High Performance Web Sites

14 Steps to Faster-Loading Web Sites

Nate Koechley 作序推荐

高性能网站建设指南

前端工程师技能精髓

Steve Souders 著

刘彦博 译

O'REILLY®

電子工業出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

http://www.phei.com.cn

大家网
TopSage.com

O'REILLY®

高性能网站建设指南

High Performance Web Sites

[美] Steve Souders 著
刘彦博 译

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

www.TopSage.com

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java一览无余：Java视频教程 | Java SE | Java EE

.Net技术精品资料下载汇总：ASP.NET篇

.Net技术精品资料下载汇总：C#语言篇

.Net技术精品资料下载汇总：VB.NET篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI脚本语言编程学习资源下载地址大全

Python语言编程学习资料(电子书+视频教程)下载汇总

最新最全Ruby、Ruby on Rails精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL篇 | SQL Server篇 | Oracle篇

平面设计优秀资源学习下载 | Flash优秀资源学习下载 | 3D动画优秀资源学习下载

最强HTML/xHTML、CSS精品学习资料下载汇总

最新JavaScript、Ajax典藏级学习资料下载分类汇总

网络最强PHP开发工具+电子书+视频教程等资料下载汇总

UML学习电子资下载汇总 软件设计与开发人员必备

经典LinuxCBT视频教程系列 Linux快速学习视频教程一帖通

天罗地网：精品Linux学习资料大收集(电子书+视频教程) Linux参考资源大系

Linux系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD精品学习资源索引 含书籍+视频

Solaris/OpenSolaris电子书、视频等精华资料下载索引

大家网
TopSage.com

内 容 简 介

本书结合 Web 2.0 以来 Web 开发领域的最新形势和特点，介绍了网站性能问题的现状、产生的原因，以及改善或解决性能问题的原则、技术技巧和最佳实践。重点关注网页的行为特征，阐释优化 Ajax、CSS、JavaScript、Flash 和图片处理等要素的技术，全面涵盖浏览器端性能问题的方方面面。在本书中，作者给出了 14 条具体的优化原则，每一条原则都配以范例佐证，并提供了在线支持。全书内容丰富，主要包括减少 HTTP 请求、Edge Computing 技术、Expires Header 技术、gzip 组件、CSS 和 JavaScript 最佳实践、主页内联、Domain 最小化、JavaScript 优化、避免重定向的技巧、删除重复 JavaScript 的技巧、关闭 ETags 的技巧、Ajax 缓存技术和最小化技术等。本书适合 Web 架构师、信息架构师、Web 开发人员及产品经理阅读和参考。

High performance Web Sites. Copyright © 2007 by O'Reilly Media, Inc.
Simplified Chinese edition, jointly published by O'Reilly Media Inc. and Publishing House of Electronics Industry, 2008. Authorized translation of the English edition, 2007 O'Reilly Media Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O'Reilly Media, Inc.授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号 图字：01-2007-5379

图书在版编目（CIP）数据

高性能网站建设指南／（美）桑德斯（Sounders,S.）著；刘彦博译.—北京：电子工业出版社，2008.6

书名原文：High Performance Web Sites

ISBN 978-7-121-06619-1

I. 高… II.①桑…②刘… III.网站－开发－指南 IV.TP393.092-62

中国版本图书馆 CIP 数据核字（2008）第 062519 号

责任编辑：何　艳
项目管理：梁　晶
印　　刷：北京天竺颖华印刷厂
装　　订：三河市金马印装有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：11　字数：215 千字
印　　次：2008 年 6 月第 1 次印刷
定　　价：35.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：(010)88258888。

www.TopSage.com

大家网
TopSage.com

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一)到GNN(最早的Internet 门户和商业网站)，再到 WebSite（第一个桌面 PC 的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

www.TopSage.com

www.TopSage.com

译者序

TopSage.com

同样的网络环境，看着别人的网站"唰"地一下就展现出来，你是否和我一样，心急如焚，盼望着早一点攒出一笔钱，给服务器加点内存？或者你已经挽起袖子，开始研究数据库优化？又或者你在暗自思量着可以把哪些设计模式或编码技巧运用在自己的后台代码里，盼望以此带来性能上的巨幅提升？

哦，别激动，很多时候事情并没有你想象的这么严重。

我们知道，一次 Web 应用程序请求，就是从浏览器发出一些参数到你的服务器，然后服务器上的程序对请求进行处理，再生成浏览器可以识别的内容（HTML、脚本、CSS、图片、Flash……），最后由浏览器将这些内容展现给访问者。人们将这一过程划分为"后端"和"前端"两个部分。

"后端"用于分析用户请求、执行数据查询并对结果进行组织，形成浏览器可以呈现的内容；"前端"负责将后端生成的内容通过网络发送给客户端浏览器。人的思维往往会进入一种误区，认为"后面的"、"背后的"东西都是神秘的、伟大的，影响力非凡。所以很多书以"某某内幕"为题，很多程序员以精通"底层开发"或"后端开发"为荣；同样的，当网站出现问题时，我们第一时间想到的也是如何优化"后端"。

本书从一开始就帮我们端正了在网站性能方面的看法，带我们走出误区。然后，从各个方面通过正例和反例的对比，让我们看到"前端"对网站性能的影响是如此巨大，而从"前端"入手改善现状是那么地简单明了。在对后端大动干戈之前，您的确应该按照本书的建议，首先从前端入手，改善性能，这样必将事半功倍。

当然，本书最大的价值在于，作者通过一系列"步骤"详细地阐明了如何通过修改前端来改善网站性能，而这些方法需要经过大量实践才能掌握并总结成文。我们应该感谢作者能够将他多年来在网站性能方面积累下来的经验总结成文，并以图书的形式分享给各位读者。而我，很荣幸能有机会将这样好的作品带给更多的中国读者。

感谢博文视点资讯有限公司的各位朋友，谢谢你们给了我这样一个机会，能把这本书带给中国读者；也感谢你们能够体谅我在翻译工作中犯下的错误和拖延的时间。感谢在网络上

www.TopSage.com

留下技术文章的英雄们，有了你们的文章内容作参考，我对术语的把握更加容易了。感谢和我志同道合的爱人，在本书的翻译过程中，你不仅照顾我、鼓励我，还帮我校对了大量稿件！

在翻译的过程中，我尽可能地仔细斟酌。但术语的使用、语言的风格等很难与原著保持精确一致，也很难满足所有人的口味，还望广大读者体谅。另外，任何一本书都可能出现错误，本书也不例外。如果您发现本书有让您不满意的地方，或者是出现了错误，除了联系出版社之外，还可以通过发邮件到 *lyb.net@gmail.com* 与我联系，或在我的博客 *http://andersliu.cnblogs.com* 留言，我将在其中为本书读者提供非官方的技术支持。

刘彦博

2008 年 4 月于北京

www.TopSage.com

Praise for *High Performance Web Sites*

大家网
TopSage.com

"If everyone would implement just 20% of Steve's guidelines, the Web would be a dramatically better place. Between this book and Steve's YSlow extension, there's really no excuse for having a sluggish web site anymore."

—Joe Hewitt, Developer of Firebug debugger and Mozilla's DOM Inspector

"Steve Souders has done a fantastic job of distilling a massive, semi-arcane art down to a set of concise, actionable, pragmatic engineering steps that will change the world of web performance."

—Eric Lawrence, Developer of the Fiddler Web Debugger, Microsoft Corporation

"As the stress and performance test lead for Zillow.com, I have been talking to all of the developers and operations folks to get them on board with the rules Steve outlined in this book, and they all ask how they can get a hold of this book. I think this should be a mandatory read for all new UE developers and performance engineers here."

—Nate Moch, *www.zillow.com*

"*High Performance Web Sites* is an essential guide for every web developer. Steve offers straightforward, useful advice for making virtually any site noticeably faster."

—Tony Chor, Group Program Manager, Internet Explorer team, Microsoft Corporation

www.TopSage.com

www.TopSage.com

目录

Table of contents

序 I

前言 III

绪言 A：前端性能的重要性 1
跟踪 Web 页面性能 1
时间花在哪了？ 3
性能黄金法则 4

绪言 B：HTTP 概述 6
压缩 7
条件 GET 请求 7
Expires 8
Keep-Alive 8
更多信息 9

第 1 章：规则 1——减少 HTTP 请求 10
图片地图 10
CSS Sprites 11
内联图片 13
合并脚本和样式表 15
小结 16

第 2 章：规则 2——使用内容发布网络 18
内容发布网络 19
节省 20

www.TopSage.com

第 3 章：规则 3——添加 Expires 头 22
Expires 头 22
Max-Age 和 mod_expires 23
空缓存 VS 完整缓存 24
不仅仅是图片 25
修订文件名 27
示例 28

第 4 章：规则 4——压缩组件 29
压缩是如何工作的 29
压缩什么 30
节省 31
配置 31
代理缓存 33
边缘情形 34
压缩的实际效果 35

第 5 章：规则 5——将样式表放在顶部 37
逐步呈现 37
sleep.cgi 38
白屏 39
无样式内容的闪烁 43
前端工程师应该做什么？ 43

第 6 章：规则 6——将脚本放在底部 45
脚本带来的问题 45
并行下载 46
脚本阻塞下载 48
最差情况：将脚本放在顶部 49
最佳情况：将脚本放在底部 49
正确地放置 50

第 7 章：规则 7——避免 CSS 表达式 51
更新表达式 52
围绕问题展开工作 52
小结 54

www.TopSage.com

大家网 TopSage.com

第 8 章：规则 8——使用外部 JavaScript 和 CSS 55
内联 VS 外置 55
典型的对比结果 58
主页 58
两全其美 59

第 9 章：规则 9——减少 DNS 查找 63
DNS 缓存和 TTL 63
浏览器的视角 66
减少 DNS 查找 68

第 10 章：规则 10——精简 JavaScript 69
精简 69
混淆 70
节省 70
示例 72
锦上添花 73

第 11 章：规则 11——避免重定向 76
重定向的类型 76
重定向是如何损伤性能的 77
重定向之外的其他选择 79

第 12 章：规则 12——移除重复脚本 85
重复脚本——确有其事 85
重复脚本损伤性能 86
避免重复脚本 87

第 13 章：规则 13——配置 ETag 89
ETag 是什么？ 89
ETag 带来的问题 91
Etag——用还是不用 93
现实世界中的 ETag 94

第 14 章：规则 14——使 Ajax 可缓存 96
Web 2.0、DHTML 和 Ajax 96
异步与即时 98
优化 Ajax 请求 99
现实世界中的 Ajax 缓存 99

www.TopSage.com

第 15 章：析构十大网站 103
页面大小、响应时间、YSlow 等级 103
如何进行测试 105
Amazon 107
AOL 110
CNN 114
eBay 116
Google 120
MSN 123
MySpace 127
Wikipedia 130
Yahoo 132
YouTube 135

索引 139

www.TopSage.com

序 Foreword

大学网 TopSage.com

你很幸运能够拿到这本书。更重要的是，你的网站用户会很幸运。Steve 在这本开天辟地的书中分享了 14 项技术，哪怕只实现了这些技术中的很少几项，你的网站也会立即变快。你的用户会感谢你。

这是为什么呢？作为一个前端工程师，你拥有巨大的能力和责任。你是用户的最后一道防线。你做出的决定直接影响着他们的体验。我相信我们大量的工作之一就是照顾用户并给他们所需要的——快速。这本书是一个创建快乐用户（和老板）的工具箱。最好的结果是，一旦恰当地使用这些技术——很多情况下，这只是一次性投入——你将长期从中获得收益。

这本书将改变你进行性能优化的方式。在 Steve 开始为我们 Yahoo!的 Platform Engineering 团队研究性能之际，我还一直认为性能主要是后端问题。但他却表明前端问题可能消耗掉整体时间的 80%。我想前端性能无非就是对图片进行优化和坚持使用外部 CSS 和 JavaScript，但您手中的这 176 页书和 14 条规则却证明实际要做的工作远不止这些。

我将他的成果应用于很多网站。发现大量已经很快的网站还可以再快将近一倍。他的方法论是可靠的，他的数据有效而且具有扩展性，他的成果是强有力的。

前端工程学这门学科还很年轻，但您手中的这本书将是这项技术趋于成熟的过程中的重要一步。通过创建更好和更快（也更具享受性）的界面和体验，我们将共同提高对 Web 的期望。

为更快的上网冲浪欢呼吧！

——Nate Koechley
高级前端工程师
Yahoo! User Interface（YUI）团队，
平台开发，Yahoo! Inc.
圣弗朗西斯科，2007 年 8 月

www.TopSage.com

大家网
TopSage.com

www.TopSage.com

前言
Preface

在八年级的时候，我在历史课上感受到工业革命的巨大威力。人们用以识别和突破制造业瓶颈的技术迷住了我。在我的印象里，最好的进步是可调整的踏板工具，它使得身高不同的工人都能轻松地够到传送带——一项简单的投资提高了加工的效率。

30 年过去了，我很乐于将本书中的最佳实践比作 19 世纪的踏板工具。这些最佳实践加强了现有流程。它们需要前期投资，但开销很小——尤其是与收益相比。而且一旦合理地运用了这些改进，它们将在整个开发过程中持续提升性能。我希望你能发现，这些用于建设高性能网站的规则能够为你和你的用户带来利益。

本书是如何组织的
How This Book Is Organized

在两章的快速介绍之后，将进入本书的主要部分——14 个性能规则。每个规则都进行了介绍，按照优先级顺序，每章一个。并非每个规则都要应用于每个网站，也不是每个网站都应该按同一种方式运用一个规则，但每个规则都值得考虑。本书的最后一章介绍了如何从性能的角度来分析 Web 页面，这一章还包含一些案例研究。

绪言 A，前端性能的重要性解释了有至少 80%的时间花在了显示 Web 页面上，而这是在 HTML 文档下载完毕后发生的，这一章还描述了本书中的技术的重要性。

绪言 B，HTTP 概述对 HTTP 进行了简要介绍，主要强调了其中与性能相关的部分。

www.TopSage.com

第 1 章，规则 1——减少 HTTP 请求介绍了为什么额外的 HTTP 请求会对性能产生巨大的影响，并介绍了减少 HTTP 请求的方法，包括图片地图、CSS Sprites、使用 `data:`模式的 URL 内联图片，以及合并脚本和样式表。

第 2 章，规则 2——使用内容发布网络强调了使用内容发布网络的优势。

第 3 章，规则 3——添加 Expires 头研究了一个简单的 HTTP 头是如何通过使用浏览器缓存来戏剧性地改善 Web 页面性能的。

第 4 章，规则 4——压缩组件解释了压缩是如何工作的，以及如何为 Web 服务器启用压缩，并讨论了现今存在的一些兼容性问题。

第 5 章，规则 5——将样式表放在顶部展示了样式表是如何影响页面呈现的。

第 6 章，规则 6——将脚本放在底部展示了脚本是如何影响呈现的，以及脚本是如何下载到浏览器中的。

第 7 章，规则 7——避免 CSS 表达式讨论了 CSS 表达式的使用和度量其影响的重要性。

第 8 章，规则 8——使用外部 JavaScript 和 CSS 介绍了如何权衡是内联 JavaScript 和 CSS 还是将它们放到外部文件中。

第 9 章，规则 9——减少 DNS 查找强调了解析域名时的频繁查找所产生的影响。

第 10 章，规则 10——精简 JavaScript 量化了从 JavaScript 中移除空白字符所带来的收益。

第 11 章，规则 11——避免重定向对使用重定向提出了警示，并给出了可替代的方法。

第 12 章，规则 12——移除重复脚本展示了如果一个页面中包含两处相同的脚本会发生什么情况。

第 13 章，规则 13——配置 ETag 介绍了 ETag 是如何工作的，以及为什么对于任何拥有多于一台 Web 服务器的网站来说，默认的实现都是不好的。

第 14 章，规则 14——使 Ajax 可缓存强调在使用 Ajax 时牢记这些性能规则的重要性。

第 15 章，析构十大网站就如何确定现实世界中的网站的性能改进给出了一些实例。

www.TopSage.com

大家网 TopSage.com

大家网 TopSage.com

本书中使用的约定
Conventions Used in This Book

以下是本书使用的排版约定：

斜体（以及黑体）

指出新术语、URL、Email 地址、文件名、文件扩展名、路径名、目录、Unix 实用工具和普通的强调。

`等宽字体`

指出广义上的计算机代码。这包括命令、选项、开关、变量、属性（Attribute）、键、函数、类型、命名空间、方法、属性（Property）、参数、值、对象、事件、事件处理器、XML 标签、HTML 标签、宏、文件内容和命令的输出。

HTTP 请求和响应以图形化方式指出，如下面的例子所示。

```
GET / HTTP 1.1 is an HTTP request header
```

```
HTTP 1.1 200 OK is an HTTP response header
```

代码示例
Code Examples

从本书配套的网站上可以找到在线示例：

http://stevesouders.com/hpws

示例包含在每章中讨论它们的上下文中。这里也列出一份，以便于查看。

无图片地图的示例（第 1 章）
http://stevesouders.com/hpws/imagemap-no.php

图片地图的示例（第 1 章）
http://stevesouders.com/hpws/imagemap.php

CSS Sprites 的示例（第 1 章）
http://stevesouders.com/hpws/sprites.php

内联图片的示例（第 1 章）
http://stevesouders.com/hpws/inline-images.php

内联 CSS 图片的示例（第 1 章）
http://stevesouders.com/hpws/inline-css-images.php

分离脚本的示例（第 1 章）
http://stevesouders.com/hpws/combo-none.php

www.TopSage.com

大家网
TopSage.com

合并脚本的示例（第 1 章）
http://stevesouders.com/hpws/combo.php

CDN 的示例（第 2 章）
http://stevesouders.com/hpws/ex-cdn.php

无 CDN 的示例（第 2 章）
http://stevesouders.com/hpws/ex-nocdn.php

无 Expires 的示例（第 3 章）
http://stevesouders.com/hpws/expiresoff.php

长久的 Expires 的示例（第 3 章）
http://stevesouders.com/hpws/expireson.php

无压缩的示例（第 4 章）
http://stevesouders.com/hpws/nogzip.html

压缩 HTML 的示例（第 4 章）
http://stevesouders.com/hpws/gzip-html.html

压缩所有组件的示例（第 4 章）
http://stevesouders.com/hpws/gzip-all.html

将 CSS 放在底部的示例（第 5 章）
http://stevesouders.com/hpws/css-bottom.php

将 CSS 放在顶部的示例（第 5 章）
http://stevesouders.com/hpws/css-top.php

将 CSS 放在顶部并使用@import 的示例（第 5 章）
http://stevesouders.com/hpws/css-top-import.php

无样式内容的 CSS 闪烁的示例（第 5 章）
http://stevesouders.com/hpws/css-fouc.php

将脚本放在中部的示例（第 6 章）
http://stevesouders.com/hpws/js-middle.php

脚本阻塞下载的示例（第 6 章）
http://stevesouders.com/hpws/js-blocking.php

将脚本放在顶部的示例（第 6 章）
http://stevesouders.com/hpws/js-top.php

将脚本放在底部的示例（第 6 章）
http://stevesouders.com/hpws/js-bottom.php

顶部脚本 VS 底部脚本的示例（第 6 章）
http://stevesouders.com/hpws/move-scripts.php

延迟脚本的示例（第 6 章）
http://stevesouders.com/hpws/js-defer.php

表达式计数器的示例（第 7 章）
http://stevesouders.com/hpws/expression-counter.php

www.TopSage.com

大家网
TopSage.com

一次性表达式的示例（第 7 章）
http://stevesouders.com/hpws/onetime-expressions.php

事件处理器的示例（第 7 章）
http://stevesouders.com/hpws/event-handler.php

内联 JS 和 CSS 的示例（第 8 章）
http://stevesouders.com/hpws/inlined.php

外部 JS 和 CSS 的示例（第 8 章）
http://stevesouders.com/hpws/external.php

可缓存的外部 JS 和 CSS 的示例（第 8 章）
http://stevesouders.com/hpws/external-cacheable.php

加载后下载的示例（第 8 章）
http://stevesouders.com/hpws/post-onload.php

动态内联的示例（第 8 章）
http://stevesouders.com/hpws/dynamic-inlining.php

一般的小脚本的示例（第 10 章）
http://stevesouders.com/hpws/js-small-normal.php

经过精简的小脚本的示例（第 10 章）
http://stevesouders.com/hpws/js-small-minify.php

经过混淆的小脚本的示例（第 10 章）
http://stevesouders.com/hpws/js-small-obfuscate.php

一般的大脚本的示例（第 10 章）
http://stevesouders.com/hpws/js-large-normal.php

经过精简的大脚本的示例（第 10 章）
http://stevesouders.com/hpws/js-large-minify.php

经过混淆的大脚本的示例（第 10 章）
http://stevesouders.com/hpws/js-large-obfuscate.php

XMLHttpRequest 信标的示例（第 11 章）
http://stevesouders.com/hpws/xhr-beacon.php

图片信标的示例（第 11 章）
http://stevesouders.com/hpws/redir-beacon.php

重复脚本——无缓存的示例（第 12 章）
http://stevesouders.com/hpws/dupe-scripts.php

重复脚本——有缓存的示例（第 12 章）
http://stevesouders.com/hpws/dupe-scripts-cached.php

重复脚本——10 次缓存的示例（第 12 章）
http://stevesouders.com/hpws/dupe-scripts-cached10.php

www.TopSage.com

一般来说，你可以在程序和文档中使用本书和在线示例中的代码。无需联系我们以求许可，除非你复制了代码中的重要部分。例如，使用本书中的大量代码来编写一个程序无需许可。销售或分发包含 O'Reilly 书中示例的光盘则需要许可。借鉴本书和引用示例代码来回答问题无需许可。将本书中的大量示例代码并入你的产品文档则需要许可。

我们重视但并不强求引用说明。引用说明通常包括标题、作者、出版社和 ISBN。例如"*High Performance Web Sites* by Steve Souder, Copyright 2007 Steve Sounder, 978-0-596-52930- 7。"

如果你感觉你对代码示例的使用超出了简单使用或前面给出的许可范围，可以通过 *permissions@oreilly.com* 免费与我们联系。

联系我们
How to Contacts Us

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下方式与我们联系。

奥莱理软件（北京）有限公司

北京市　西城区　西直门南大街 2 号　成铭大厦 C 座 807
邮政编码：100035
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

与本书有关的在线信息如下所示。

http://www.oreilly.com/catalog/9780596529307（原书）
http://www.oreilly.com.cn/book.php?bn=978-7-121-06619-1（中文版）

www.TopSage.com

大家网 TopSage.com

感谢
Acknowledgments

Ash Patel 和 Geoff Ralston 是 Yahoo!的执行人员，他们让我启动了一个中心，专门研究性能。很多 Yahoo!人帮忙回答问题并讨论了观点——Ryan Troll、Doug Crockford、Nate Koechley、Mark Nottingham、Cal Henderson、Don Vail 和 Tenni Theurer。我的编辑 Andy Oram 付出了极大的耐心，并给我这个第一次当作者的人以必要的鼓励。很多人帮助检查了这本书——Doug Crockford、Havi Hoffman、Cal Henderson、Don Knuth，尤其是 Jeffrey Friedl、Alexander Kirk 和 Eric Lawrence。

本书完全是在周末和深夜的业余时间中完成的。感谢我的妻子和女儿在周末给我这些时间来工作。感谢我的父母教授我在深夜工作需遵循的道德规范。

www.TopSage.com

大家网
TopSage.com

联系博文视点

您可以通过如下方式与本书的出版方取得联系。

读者信箱：reader@broadview.com.cn

投稿信箱：bvtougao@gmail.com

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室

邮政编码：430074

电话：（027）87690813　　传真：（027）87690813 转 817

若您希望参加博文视点的有奖读者调查，或对写作和翻译感兴趣，欢迎您访问：
http://bv.csdn.net

关于本书的勘误、资源下载及博文视点的最新书讯，欢迎您访问博文视点官方博客：
http://blog.csdn.net/bvbook

www.TopSage.com

前端性能的重要性
The Importance of Frontend Performance

我的 Web 职业生涯中的大部分时间都是担任后端工程师。因此，我一直很忠实地实现性能设计、进行正规的后端优化——编译器选项、数据库索引和内存管理等。很多书都关注于如何在这些领域中进行优化，在寻求改进的时候，大量的时间也都花在这些地方。事实上，只有 10%~20%的最终用户响应时间是花在从 Web 服务器获取 HTML 文档并传送到浏览器中的。如果希望能够有效地减少页面的响应时间，就必须关注剩余 80%~90%的最终用户体验。这 80%~90%的时间花在哪里了？如何减少它？后面的章节将介绍一些基础知识，用于理解今天的 Web 页面，并给出了使它们变得更快的 14 条规则。

跟踪 Web 页面性能
Tracking Web Page Performance

为了知道能够改进哪些地方，我们需要了解用户的时间都花在等待哪些东西上了。图 A-1 展示了当使用 Internet Explorer 下载 Yahoo!的首页（*http://www.yahoo.com*）时产生的 HTTP 流量。每个横条都是一个 HTTP 请求。第一个横条，标有 `html`，是对 HTML 文档的初始请求。浏览器会解析 HTML 并开始下载页面中的组件。在这种情况下，浏览器的缓存是空的，因此必须下载所有的组件。HTML 文档只占总响应时间的 5%。用户需要花费其余 95%的时间中的大部分来等待组件的下载。还有一小部分时间花在解析 HTML、脚本和样式表上面，从下载横条的空白间隙可以看出这一点。

图 A-2 展示了在 IE 中第二次下载同一个 URL 的情况。HTML 文档只占了总响应时间的 12%。很多组件无需下载，因为它们已经存在于浏览器的缓存中了。

www.TopSage.com

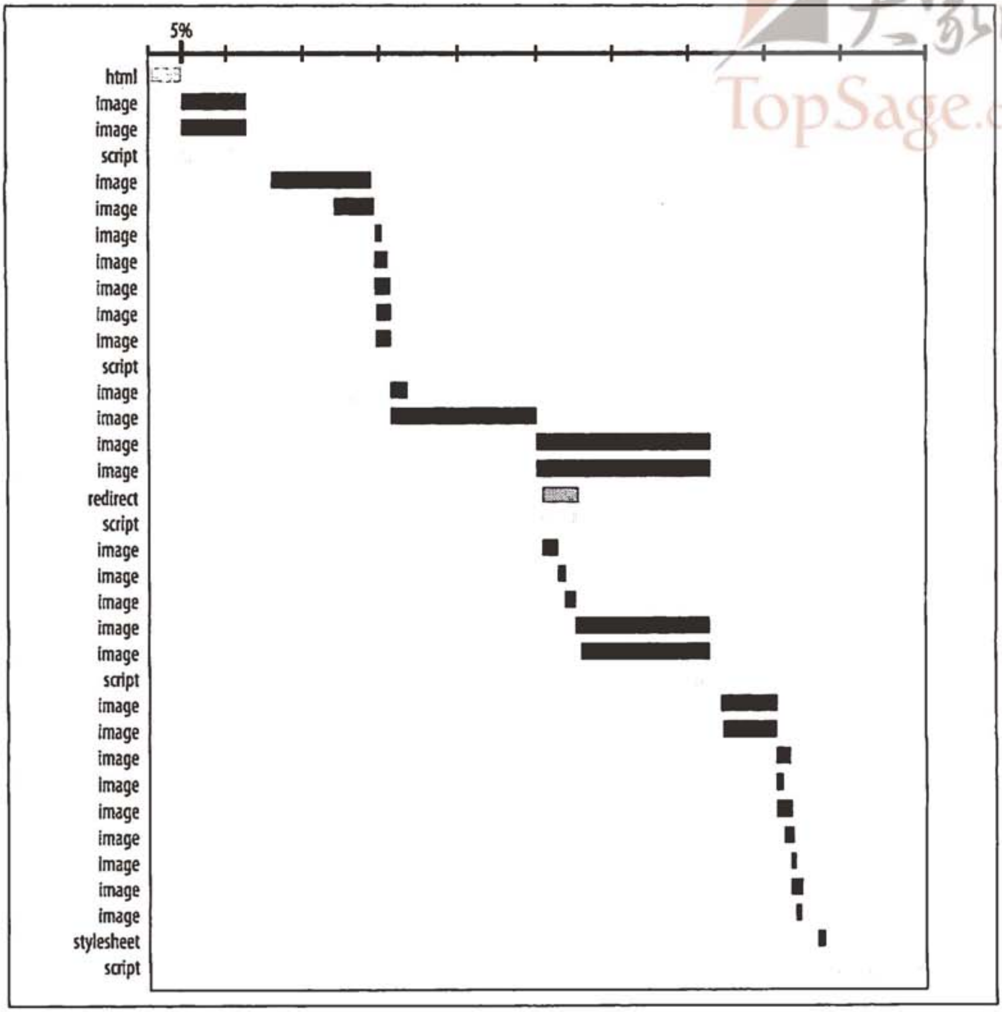


图 A-1：在 IE 中下载 http://www.yahoo.com，缓存为空

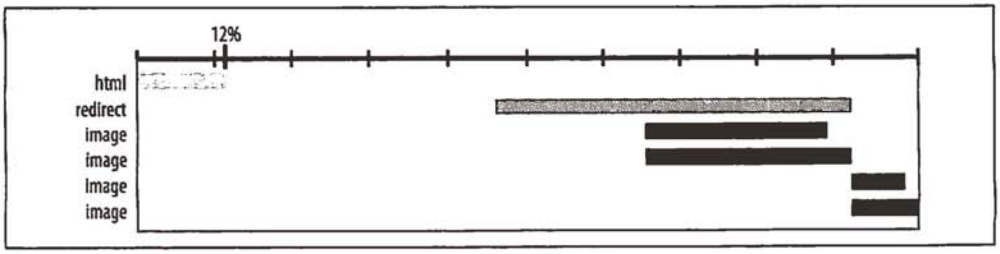


图 A-2：在 IE 中下载 http://www.yahoo.com，完整缓存

www.TopSage.com

大家网 TopSage.com

在第二次页面查看时仍然有五个组件需要重新请求：

一个重定向

这个重定向之前已经下载过了，但浏览器需要再一次请求它。HTTP 响应的状态码是 302（Found 或 Moved Temporarily），并且在响应头中没有缓存信息，因此浏览器无法缓存该响应。我们将在绪言 B 中深入讨论 HTTP。

三个未缓存的图片

接下来的三个请求用于初次页面查看时未下载的图片。这些图片用于经常变化的新闻照片和广告。

一个缓存的图片

最后的 HTTP 请求是一个**条件 *GET* 请求**（*Conditional GET request*）。该图片已经被缓存，但由于这个 HTTP 响应头的存在，浏览器必须在将这幅图片显示给用户之前对其检查更新。条件 GET 请求也会在绪言 B 中进行介绍。

时间花在哪了？
Where Does the Time Go?

以这种方式查看了 HTTP 流量后，很容易看出时间花在了哪里。至少 80%的最终用户响应时间花在了页面中的组件上。如果我们继续深入挖掘这些图表的细节，将开始看到浏览器和 HTTP 之间的相互影响是多么的复杂。我们已经讨论了 HTTP 状态码和响应头是如何影响浏览器缓存的。此外，我们还可以进行以下观察。

- 有缓存的场景（图 A-2）并没有太多的下载活动。可以看到紧跟 HTML 文档的 HTTP 请求之后的是一段空白，没有进行下载。在这段时间里，浏览器正在解析 HTML、JavaScript 和 CSS，并从缓存中获取组件。
- 大量的 HTTP 请求并行发生。图 A-2 中最大的三个 HTTP 请求是并行发生的，而在图 A-1 中有多达六或七个并发的 HTTP 请求，不管是使用 HTTP 1.0 还是 1.1。这一行为是因为使用了多个不同的主机名造成的，第 6 章的“并行下载”一节将解释这些问题。
- 在请求脚本时不会发生并行请求。这是因为在很多情况下，浏览器在下载脚本时会阻塞额外的 HTTP 请求。第 6 章解释了为什么会这样，以及如何利用这一信息来改善页面的加载时间。

要精确地指出时间花在哪里是很有挑战性的工作。但很容易看出时间**没有**花在哪里——它**没有**花在下载 HTML 文档上，包括任何的后端处理。这就是为什么前端性能很重要。

www.TopSage.com

大家网
TopSage.com

性能黄金法则
The Performance Golden Rule

仅需要花费 10%~20%的响应时间来下载 HTML 文档这一现象并不只出现在 Yahoo!的首页上。这一统计数字适用于我所分析过的所有的 Yahoo!功能（除了 Yahoo! Search，因为它的页面上只有很少量的组件）。而且，这一统计数字适用于绝大多数网站。表 A-1 展示了从 *http://www.alexa.com* 上得到的前十个美国网站。注意除了 AOL 之外，这些网站都名列美国前十名。Craigslist.org 也位于前十名，但它的页面中几乎没有图片、脚本和样式表，并不是一个很好的实例。因此我在这里选择用 AOL 来代替它。

表 A-1：十大网站花在下载 HTML 文档上的时间百分比

	无缓存	完整缓存
AOL	6%	14%
Amazon	18%	14%
CNN	19%	8%
eBay	2%	8%
Google	14%	36%
MSN	3%	5%
MySpace	4%	14%
Wikipedia	20%	12%
Yahoo!	5%	12%
YouTube	3%	5%

所有这些网站在获取 HTML 文档时，花费的时间都不到总响应时间的 20%。其中一个例外是 Google 在完整缓存场景中的情况。这是因为 *http://www.google.com* 只有 6 个组件，除了其中一个之外，都被配置为可以由浏览器进行缓存。在后续的页面查看过程中，所有这些组件都已被缓存，只需对 HTML 文档和一个图片信标进行 HTTP 请求。

在进行优化时，关键是剖析当前的性能，找到在哪里能够获得最大的改进。很明显，在这种情况下我们应该关注前端性能。

首先，关注前端可以很好地提高整体性能。如果我们可以将后端响应时间缩短一半，整体响应时间只能减少 5%~10%。而如果关注前端性能，同样是将其响应时间减少一半，则整体响应时间可以减少 40%~45%。

www.TopSage.com

其次，改进前端通常只需要较少的时间和资源。减少后端延迟会带来很大的改动，例如重新设计应用程序的架构和代码、查找和优化临界代码路径、添加或改动硬件、对数据库进行分布化等。这些改动需要花费数周或数月。接下来的章节中将要介绍的前端性能改进只需要一些最佳实践，例如修改 Web 服务器配置文件（第 3 章和第 4 章）、将脚本和样式表放在页面中的特定位置（第 5 章和第 6 章）、合并图片、脚本和样式表（第 1 章）。这些改动只需要几个小时或几天，这比进行后端改进要少花很多时间。

第三，前端性能调整已被证明是可行的。Yahoo!中有超过 50 个团队使用了这里介绍的最佳实践并降低了最终用户响应时间，降低的幅度通常为 25%或更高。有的时候，我们必须超越这些规则，并根据对网站的分析进行更有针对性的改进。但一般来说，只需要遵守这些最佳实践就能节省 25%或更多的时间。

在开始任何新的性能改善计划之前，我绘制了一个类似表 A-1 的图表，并解释一下**性能黄金法则**：

> 只有 10%~20%的最终用户响应时间花在了下载 HTML 文档上。其余的 80%~90%时间花在了下载页面中的所有组件上。

本书的其余部分对于减少 80%~90%的最终用户响应时间给出了精确的指导。为了证实这一点，我将涵盖跨度很大的技术——HTTP 头、JavaScript、CSS、Apache 等。

因为 HTTP 的基本知识有助于理解本书，我将在绪言 B 中对这些基本知识进行介绍。

在这之后是 14 条提升性能的规则，每章介绍一个。这些规则按照常规的优先级顺序列出。特定的网站，规则的适用性可能会不同。例如，规则 2 更适用于商业网站，而对个人网站就不可行了。如果你遵从所有适用于你的网站的规则，你的页面的加载速度会提高 20%~25%，用户体验也将得到改善。本书的最后一部分展示了如何从性能的角度去分析十大美国网站。

www.TopSage.com

大家网 TopSage.com

绪言 B

HTTP 概述

HTTP Overview

在介绍使 Web 页面加载速度更快的具体规则之前，有必要理解部分 Hyper Text Transfer Protocol（HTTP）对性能的影响。HTTP 是浏览器和服务器通过 Internet 进行相互通信的协议。HTTP 规范由 World Wide Web Consortium（W3C）和 Internet Engineering Task Force（IETF）进行编制，文档是 RFC 2616。HTTP 1.1 是今天比较常见的版本，但一些浏览器和服务器还在使用 HTTP 1.0。

HTTP 是一种客户端/服务器协议，由请求和响应构成。浏览器向一个特定的 URL 发送 HTTP 请求，URL 对应的宿主服务器发回 HTTP 响应。和很多 Internet 服务一样，该协议使用简单的纯文本格式。请求的类型有 GET、POST、HEAD、PUT、DELETE、OPTIONS 和 TRACE。我们主要关注最常见的 GET 请求。

GET 请求包含一个 URL，然后是头。HTTP 响应包含状态码、头和响应体。下面的例子展示了当请求脚本 *yahoo_2.0.0-b2.js* 时可能产生的 HTTP 头。

```
GET /us.js.yimg.com/lib/common/utils/2/yahoo_2.0.0-b2.js
HTTP 1.1
Host: us.js2.yimg.com
User-Agent: Mozilla/5.0(...) Gecko/20061206 Firefox/1.5.0.9
```

```
HTTP 1.1 200 OK
Content-Type: application/x-javascript
Last-Modified: Wed, 22 Feb 2006 04:15:54 GMT
Content-Length: 355

var YAHOO=...
```


www.TopSage.com

大家网 TopSage.com

压缩
Compression

如果浏览器和服务器都支持的话，可以使用压缩来减小响应的大小。浏览器可以使用 Accept-Encoding 头来声明它支持压缩。服务器使用 Content-Encoding 头确认响应已被压缩。

```
GET /us.js.yimg.com/lib/common/utils/2/yahoo_2.0.0-b2.js
HTTP 1.1
Host: us.js2.yimg.com
User-Agent: Mozilla/5.0(...) Gecko/20061206 Firefox/1.5.0.9
Accept-Encoding: gzip,deflate
```

```
HTTP 1.1 200 OK
Content-Type: application/x-javascript
Last-Modified: Wed, 22 Feb 2006 04:15:54 GMT
Content-Length: 255
Content-Encoding: gzip

^_\213^H^@^@^@^@^@^@^Cl\217\315j\3030^P\204_E\361IJ...
```

注意响应体是如何压缩的。第 4 章解释了如何启用压缩，并对由于代理缓存导致的边缘情形给出了警告。这一章还讨论了 Vary 和 Cache-Control 头。

条件 GET 请求
Conditional GET Requests

如果浏览器在其缓存中保留了组件的一个副本，但并不确定它是否仍然有效，就会生成一个条件 GET 请求。如果确认缓存的副本仍然有效，浏览器就可以使用缓存中的副本，这会得到更小的响应和更快的用户体验。

典型情况下，缓存副本的有效性源自其最后修改时间。基于响应中的 Last-Modified 头（参见前面的示例），浏览器可以知道组件最后的修改时间。它会使用 If-Modified-Since 头将最后修改时间发送给服务器。浏览器其实是在说："我拥有这个组件的一个版本，这是它的最后修改时间。我可以使用它吗？"

```
GET /us.js.yimg.com/lib/common/utils/2/yahoo_2.0.0-b2.js
HTTP 1.1
Host: us.js2.yimg.com
User-Agent: Mozilla/5.0(...) Gecko/20061206 Firefox/1.5.0.9
Accept-Encoding: gzip,deflate
If-Modified-Since: Wed, 22 Feb 2006 04:15:54 GMT
```

```
HTTP 1.1 304 Not Modified
Content-Type: application/x-javascript
Last-Modified: Wed, 22 Feb 2006 04:15:54 GMT
```


www.TopSage.com

如果组件自生成日期以来没有改变过，服务器会返回一个“304 Not Modified”状态码并不再发送响应体，从而得到一个更小且更快的响应。在 HTTP 1.1 中，ETag 和 If-None-Match 头是进行条件 GET 请求的另外一种方式。这两种方式都将在第 13 章中进行讨论。

大家网 TopSage.com

Expires

条件 GET 请求和 304 响应有助于让页面加载得更快，但仍需要在客户端和服务器之间进行一次往返确认，以执行有效性检查。Expires 头通过明确指出浏览器是否可以使用组件的缓存副本来消除这个需要。

```
HTTP 1.1 200 OK
Content-Type: application/x-javascript
Last-Modified: Wed, 22 Feb 2006 04:15:54 GMT
Expires: Wed, 05 Oct 2016 19:16:20 GMT
```

当浏览器看到响应中有一个 Expires 头时，它会和相应的过期时间组件一起保存到其缓存中。只要组件没有过期，浏览器就会使用缓存版本而不会进行任何 HTTP 请求。第 3 章详细地讲述了 Expires 和 Cache-Control 头。

Keep-Alive

HTTP 构建在 Transmission Control Protocol（TCP）之上。在 HTTP 的早期实现中，每个 HTTP 请求都要打开一个 socket 连接。这样做效率很低，因为一个 Web 页面中的很多 HTTP 请求都指向同一个服务器。例如，很多为 Web 页面中的图片发起的请求都指向一个通用的图片服务器。**持久连接**（*Persistent Connection*）的引入解决了多对一请求服务器导致的 socket 连接低效性的问题。它使浏览器可以在一个单独的连接上进行多个请求。浏览器和服务器使用 Connection 头来指出对 Keep-Alive 的支持。在服务器的响应中 Connection 头看起来是一样的。

```
GET /us.js.yimg.com/lib/common/utils/2/yahoo_2.0.0-b2.js
HTTP 1.1
Host: us.js2.yimg.com
User-Agent: Mozilla/5.0(...) Gecko/20061206 Firefox/1.5.0.9
Accept-Encoding: gzip,deflate
Connection: keep-alive
```

```
HTTP 1.1 200 OK
Content-Type: application/x-javascript
Last-Modified: Wed, 22 Feb 2006 04:15:54 GMT
Connection: keep-alive
```


www.TopSage.com

浏览器或服务器可以通过发送一个 `Connection: close` 头来关闭连接。从技术上讲，`Connection: keep-alive` 并不是 HTTP 1.1 中必需的，但很多浏览器和服务器都包含它。

HTTP 1.1 中定义的管道可以在一个单独的 socket 上发送多个请求而无须等待响应。管道的性能要优于持久连接。但不幸的是，Internet Explorer（包括版本 7 在内）都不支持管道，而 Firefox 自从版本 2 开始默认也是关闭该功能的。在管道被广泛应用之前，Keep-Alive 依然是浏览器和服务器使用 HTTP 的 socket 连接最有效的方式。这对于 HTTPS 来说甚至更重要，因为建立新的安全 socket 连接要消耗更多的时间。

更多信息
There's More

本绪言对 HTTP 进行的介绍仅仅是一个概述，并且只关注其影响性能的方面。如果要进行深入学习，可以阅读 HTTP 规范（*http://www.w3.org/Protocols/rfc2616/rfc2616.html*）和 David Gourley、Brian Totty 编著的《HTTP: The Definitive Guide》一书（O'Reilly，*http://www.oreilly.com/catalog/httptdg*）。这里所强调的部分知识已经足够用于理解后面章节所介绍的最佳实践了。

www.TopSage.com

第 1 章

规则 1——减少 HTTP 请求

Rule 1: Make Fewer HTTP Requests

绪言 A 中介绍的**性能黄金法则**（*Performance Golden Rule*）揭示了只有 10%~20%的最终用户响应时间花在接收所请求的 HTML 文档上。剩下的 80%~90%时间花在为 HTML 文档所引用的所有组件（图片、脚本、样式表、Flash 等）进行的 HTTP 请求上。因此，改善响应时间的最简单途径就是减少组件的数量，并由此减少 HTTP 请求的数量。

从页面中移除组件的想法会引发性能和产品设计之间的矛盾。在这一章中，我将介绍的技术既可以减少 HTTP 请求，又能避免在性能和设计之间进行艰难的抉择。这些技术包括图片地图、CSS Sprites、内联图片和脚本、样式表的合并。运用这些技术在示例页面上估计响应时间减少到 50%左右。

图片地图

Image Maps

在一个最简单的转筒中，超链接带有一些文本，并被关联到目标 URL 上。一种更为美观的选择是将超链接关联到图片上，例如在导航栏或按钮中。如果是以这种形式关联多个带有超链接的图片，使用图片地图这种方式就既能减少 HTTP 请求，又无需改变页面外观感受。图片地图（*Image Map*）允许你在一个图片上关联多个 URL。目标 URL 的选择取决于用户单击了图片上的哪个位置。

图 1-1 给出了一个示例，在一个导航栏上有五幅图片。单击一个图片会将你带到与之相关的链接。这可以通过五个分开的超链接、使用五个分开的图片来实现。然而，如果使用一个图片地图则可以更有效率，因为五个 HTTP 请求被减少为只有一个 HTTP 请求。响应时间将会降低，因为减少了 HTTP 开销。

你可以通过访问下面的 URL 来自己试验一下。单击每个链接来看一下获取时间。

www.TopSage.com

大家网 TopSage.com

图 1-1：图片地图示例

无图片地图的示例
　　http://stevesouders.com/hpws/imagemap-no.php

图片地图的示例
　　http://stevesouders.com/hpws/imagemap.php

当在 DSL（900Kbps）上使用 Internet Explorer 6.0 时，获取图片地图的时间比获取为每个超链接使用分离图片的导航条的时间快 56%（354ms : 799ms）。这是因为图片地图减少了四个 HTTP 请求。

图片地图有两种类型。**服务器端图片地图**（*Server-side image maps*）将所有点击提交到同一个目标 URL，向其传递用户单击的 x、y 坐标。Web 应用程序将该 x、y 坐标映射为适当的操作。**客户端图片地图**（*Client-side image maps*）更加典型，因为它可以将用户的点击映射到一个操作，而无需向后端应用程序发送请求。映射通过 HTML 的 MAP 标签实现。下面的 HTML 将图 1-1 中的导航栏转换成了图片地图，并展示了如何使用 MAP 标签。

```
<img usemap="#map1" border=0 src="/images/imagemap.gif?t=1196816255">
<map name="map1">
  <area shape="rect" coords="0,0,31,31" href="home.html" title="Home">
  <area shape="rect" coords="36,0,66,31" href="gifts.html" title="Gifts">
  <area shape="rect" coords="71,0,101,31" href="cart" title="Cart">
  <area shape="rect" coords="106,0,136,31" href="settings.html" title="Settings">
  <area shape="rect" coords="141,0,171,31" href="help.html" title="Help">
</map>
```

使用图片地图也有缺点。在定义图片地图上的区域坐标时，如果采取手工的方式则很难完成且容易出错，而且除了矩形之外几乎无法定义其他形状。通过 DHTML 创建的图片地图则在 Internet Explorer 中无法工作。

如果你正在导航栏或其他超链接中使用多个图片，将它们转换为图片地图是加速页面的最简单的方式。

CSS Sprites

和图片地图一样，*CSS Sprites* 也可以合并图片，但更为灵活。这个概念就像是使用“显灵板（Ouija Board）”一样，占卜写板（Planchette，由所有参与者一起托着的一个观察用具）不停地移动，停留在不同的字母上。要使用 CSS Sprites，需要将多个图片合并到一个单独的图片中，就像图 1-2 所展示的那样。这就是“显灵板”。

www.TopSage.com

大家网 TopSage.com

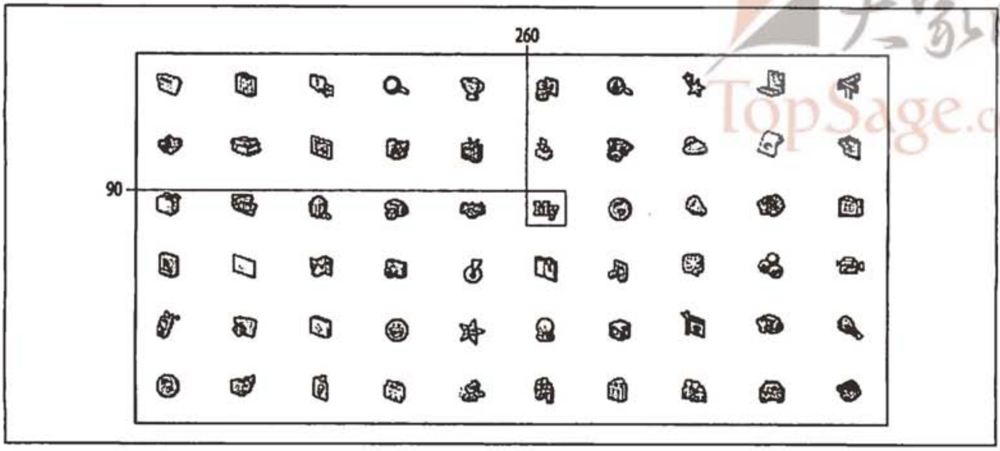


图 1-2：CSS Sprites 将多幅图片合并为一幅单独的图片

"占卜写板"是任何支持背景图片的HTML元素，如SPAN或DIV。使用CSS的background-position 属性，可以将 HTML 元素放置到背景图片中期望的位置上。例如，像下面这样可以将"My"图标用作一个元素的背景图片：

```
<div style="background-image: url('a_lot_of_sprites.gif');
            background-position: -260px -90px;
            width: 26px; height: 24px;">
</div>
```

我使用CSS Sprites修改了前面的图片地图示例。一个名为navbar的DIV包含了五个链接。每个链接都被包围在一个SPAN中，它们使用同一个背景图片——spritebg.gif——定义在#navbar span 规则中。每个 SPAN 都具有一个不同的类，通过 background-position 属性指定了 CSS Sprites 的偏移量：

```
<style>
#navbar span {
  width:31px;
  height:31px;
  display:inline;
  float:left;
  background-image:url(/images/spritebg.gif);
}
.home     { background-position:0 0; margin-right:4px; margin-left: 4px;}
.gifts    { background-position:-32px 0; margin-right:4px;}
.cart     { background-position:-64px 0; margin-right:4px;}
.settings { background-position:-96px 0; margin-right:4px;}
.help     { background-position:-128px 0; margin-right:0px;}
</style>

<div id="navbar" style="background-color: #F4F5EB; border: 2px ridge #333; width:
180px; height: 32px; padding: 4px 0 4px 0;">
```


www.TopSage.com

```
    <a href="javascript:alert('Home')" title="Home"><span class="home"></span></a>
    <a href="javascript:alert('Gifts')" title="Gifts"><span class="gifts"></span></a>
    <a href="javascript:alert('Cart')" title="Cart"><span class="cart"></span></a>
    <a href="javascript:alert('Settings')" title="Settings"><span class="settings"></span></a>
    <a href="javascript:alert('Help')" title="Help"><span class="help"></span></a>
</div>
```

它和图片地图示例几乎一样快——两者分别是342毫秒和354毫秒，其间的差别微乎其微。重要的是，它比使用分离的图片快57%。

CSS Sprites 示例

http://stevesouders.com/examples/sprites.php

但是，图片地图中的图片必须是连续的，而CSS Sprites则没有这个限制。关于CSS Sprites的赞成意见（与一些反对意见），Dave Shea在其权威作品《CSS Sprites: Image Slicing's Kiss of Death》中给出了很好的解释。我在前面简要地提到了它的优点——通过合并图片减少HTTP请求，并且比图片地图更灵活。一个令人惊奇的优点是，它还降低了下载量。很多人会认为合并后的图片会比分离的图片的总和要大，因为合并的图片中包含有附加的空白区域。实际上，合并后的图片会比分离的图片的总和要小，这是因为它降低了图片自身的开销（颜色表、格式信息，等等）。

如果需要在页面中为背景、按钮、导航栏、链接等提供大量图片，CSS Sprites绝对是一种优秀的解决方案——干净的标签、很少的图片和很短的响应时间。

内联图片

Inline Images

通过使用`data:` URL模式可以在Web页面中包含图片但无需任何额外的HTTP请求。尽管Internet Explorer目前还不支持这种方式，但它能给其他浏览器带来的节省使得它值得关注。

我们都很熟悉包含`http:`模式的URL。其他类似的模式包括`ftp:`、`file:`和`mailto:`。但除此之外还有很多模式，如`smtp:`、`pop:`、`dns:`、`whois:`、`finger:`、`daytime:`、`news:`和`urn:`。这其中有一些是官方注册的，还有一些由于广泛使用而被接受。

`data:` URL模式在1995年被首次提议。规范（*http://tools.ietf.org/html/rfc2397*）对它的描述为："允许将小块数据内联为'立即（immediate）'数"。数据就在其URL自身之中，其格式如下：

```
data:[<mediatype>][;base64],<data>
```


www.TopSage.com

一个红色五角星形状的内联图片可以定义为：

```
<IMG ALT="Red Star"
SRC="data:image/gif;base64,R0lGODlhDAAMALMLAPN8ffBiYvWW
lvrKy/FvcPewsO9VVfajo+w6O/zl5estLv/8/AAAAAAAAAAAAAAACH5BAEA
AAsALAAAAAAMAAwAAAQzcElZyryTEHyTUgknHd9xGV+qKsYirKkwDYiKDBia
tt2H1KBLQRFIJAIKywRgmhwAIlEEADs=">
```

我见过的 data: 都是用于内联图片的，但它可以用在任何需要指定 URL 的地方，包括 SCRIPT 和 A 标签。

data: URL 模式的主要缺陷在于不受 IE 的支持（直到包括版本 7 都是如此）。另一个缺陷是可能存在数据大小上的限制，但 Firefox 1.5 可以支持高达 100KB 的内联图片。Base64 编码会增加图片的大小，因此整体下载量会增加。

下面的示例将使用内联图片来实现前面的导航栏。

内联图片的示例

http://stevesouders.com/examples/inline-images.php

由于 data: URL 是内联在页面中的，在跨越不同页面时不会被缓存。不要去内联公司 Logo，因为编码过的 Logo 会导致页面变大。这种情况下，聪明的做法是使用 CSS 并将内联图片作为背景。将该 CSS 规则放在外部样式表中，这意味着数据可以缓存在样式表内部。在下面的例子中，导航栏中的每个链接所使用的背景图片都被实现为外部样式表中的内联图片。

内联 CSS 图片的示例

http://stevesouders.com/examples/inline-css-images.php

在外部样式表中，每个 SPAN 都有一个规则，其中包含了内联的背景图片：

```
.home { background-image: url(data:image/gif;base64,R0lGODlhHwAfAPcAAAAAAIxKA...);}
.gift { background-image: url(data:image/gif;base64,R0lGODlhHwAfAPcAAAAAAABCp...);}
.cart { background-image: url(data:image/gif;base64,R0lGODlhHwAfAPcAAAAAADlCr...);}
.settings { background-image: url(data:image/gif;base64,R0lGODlhHwAfAPcAAAAAA...);}
.help { background-image: url(data:image/gif;base64,R0lGODlhHwAfAPcAAAAAALW1t...);}
```

PHP 函数 file_get_contents 可以很容易地通过从磁盘中读取图片并将其内容插入到页面中来创建内联图片。在这个示例中，外部样式表的 URL 指向一个 PHP 模板——*http://stevesouders.com/hpws/inline-css-images-css.php*。这个 PHP 模板展示了 file_get_contents 的使用，它生成了前面给出的样式表：

```
.home { background-image: url(data:image/gif;base64,
   <?php echo base64_encode(file_get_contents("../images/home.gif")) ?>);}
.gift { background-image: url(data:image/gif;base64,
   <?php echo base64_encode(file_get_contents("../images/gift.gif")) ?>);}
```


www.TopSage.com

```
.cart { background-image: url(data:image/gif;base64,
  <?php echo base64_encode(file_get_contents("../images/cart.gif")) ?>);}
.settings { background-image: url(data:image/gif;base64,
  <?php echo base64_encode(file_get_contents("../images/settings.gif")) ?>);}
.help { background-image: url(data:image/gif;base64,
  <?php echo base64_encode(file_get_contents("../images/help.gif")) ?>);}
```

将这个例子与之前的示例进行比较，可以看到它和图片地图及 CSS Sprites 的响应时间几乎一样，也是比原来为每个链接使用单独的图片的方式快 50%以上。将内联图片放置在外部样式表中增加了一个额外的 HTTP 请求，但被缓存后可以得到额外的收获。

合并脚本和样式表

Combined Scripts and Stylesheets

今天的很多网站都使用了 JavaScript 和 CSS。前端工程师必须选择是对 JavaScript 和 CSS 进行“内联”（也就是将其嵌在 HTML 文档中）还是将其放在外部的脚本和样式表文件中。一般来说，使用外部脚本和样式表对性能更有利（这将在第 8 章中详细讨论）。然而，如果遵循软件工程师所推荐的方式和模块化的原则将代码分开放到多个小文件中，会降低性能，因为每个文件都会导致一个额外的 HTTP 请求。

表 1-1 表明，前十位的网站在其首页上平均使用六到七个脚本和一到二个样式表。绪言 A 中曾介绍过，这些网站选自 *http://www.alexa.com*。如果它们没有被缓存到用户的浏览器中，则每个文件都需要一个额外的 HTTP 请求。和图片地图及 CSS Sprites 的优点一样，将这些单独的文件合并到一个文件中，可以减少 HTTP 请求的数量并缩短最终用户响应时间。

表 1-1：十大网站的脚本和样式表的数量

网站	脚本	样式表
http://www.amazon.com	3	1
http://www.aol.com	18	1
http://www.cnn.com	11	2
http://www.ebay.com	7	2
http://froogle.google.com	1	1
http://www.msn.com	9	1
http://www.myspace.com	2	2
http://www.wikipedia.org	3	1
http://www.yahoo.com	4	1
http://www.youtube.com	7	3

www.TopSage.com

为了清晰，我不建议将脚本和样式表合并在一起。但是多个脚本应该合并为一个脚本，多个样式表也应该合并为一个样式表。理想情况下，一个页面应该使用不多于一个的脚本和样式表。

下面的例子展示了合并脚本是如何缩短最终用户响应时间的。使用了合并脚本的页面在加载时快了 38%。合并样式表可以带来类似的性能改进。在这一部分的剩余内容中，我将只介绍脚本（因为其使用量很大），但所有这些讨论也都同样适用于样式表。

分离脚本的示例

> *http://stevesouders.com/examples/combo-none.php*

合并脚本的示例

> *http://stevesouders.com/examples/combo.php*

对于那些接受过编写模块化代码（不论是 JavaScript 还是其他编程语言）的开发者来说，将所有东西合并到一个单独的文件中看起来像是一种倒退，而且将所有的 JavaScript 合并为一个单独的文件在开发环境中很难完成。一个页面可能需要 `script1`、`script2` 和 `script3`，而另一个页面可能需要 `script1`、`script3`、`script4` 和 `script5`。解决的方法是遵守编译型语言的模式，保持 JavaScript 的模块化，而在生成过程中从一组特定的模块生成一个目标文件。

很容易想象包含合并脚本和样式表的生成过程——简单地将适当的文件连接为一个单独的文件。合并文件很容易。在这一步中还可以对文件进行精简（参见第 10 章）。难的是组合的数量的增长。如果有大量需要不同模块的页面，组合后的数量就会非常庞大。十个脚本就能产生上千种组合！更不要指望强制每个页面都使用每个模块，而不管它是否真的需要。以我的经验来看，拥有多个页面的网站肯定会有大量不同的模块组合。这值得花一些时间去分析一下你的页面，确保组合的数量是可管理的。

小结
Conclusion

这一章介绍了我们在 Yahoo!中使用的用于减少 Web 页面中 HTTP 请求数量，而又无需在页面设计上作出妥协的技术。后面将要介绍的规则也提到了有助于减少 HTTP 请求数量的方法，但它们主要关注的是后续的页面浏览。对于那些在最初呈现页面时并非必需的组件来说，第 8 章所介绍的加载后下载（*post-onload download*）技术有助于将这些 HTTP 请求推迟到页面加载完毕后进行。

www.TopSage.com

本章的规则是在用户第一次访问你的网站时能更有效地减少 HTTP 请求的数量，这也是为什么我将它放在了第 1 章、为什么它是最重要的规则。遵守该规则可以同时改善首次浏览和后续浏览的网站响应时间。首次访问页面时的响应时间决定着用户是放弃你的网站还是不停地进行回访。

减少 HTTP 请求

www.TopSage.com

大家网
TopSage.com

第 2 章

规则 2——使用内容发布网络
Rule 2: Use a Content Delivery Network

用户平均的带宽每年都在增长，但用户对你的 Web 服务器的亲近程度仍然受到页面响应时间的影响。网站最初通常将其所有的服务器放在同一个地方。当用户群增加时，公司就必须面对服务器放置地点不再适用的事实——有必要在多个地理位置不同的服务器上部署内容。

作为实现地理位置分离的第一步，不要尝试使用分布式架构重新设计你的 Web 应用程序。这样的应用程序决定了重新设计将带来令人恐惧的任务，如同步会话状态和在服务器放置地点之间复制数据库事务。重新设计这一步骤会推迟——甚至根本无法实现——缩短用户和你的内容之间的距离这一愿望。

应用绪言 A 中提及的性能黄金法则（*Performance Golden Rule*）介绍了正确部署服务器内容的第一步：

> 只有 10%~20%的最终用户响应时间花在了下载 HTML 文档上。其余的 80%~90%时间花在了下载页面中的所有组件上。

如果应用程序 *Web* 服务器（*Application Web Server*）离用户更近，则一个 HTTP 请求的响应时间将缩短。另一方面，如果组件 *Web* 服务器（*Component Web Server*）离用户更近，则多个 HTTP 请求的响应时间将缩短。与其开始重新设计应用程序这一艰难任务，以便将应用程序 Web 服务器分散开，不如首先将组件 Web 服务器分散开。这不仅能达到响应时间大幅减少的目的，还很容易实现。感谢内容发布网络（*Content Delivery Network*）。

www.TopSage.com

大家网
TopSage.com

内容发布网络

Content Delivery Networks

内容发布网络（CDN）是一组分布在多个不同地理位置的 Web 服务器，用于更加有效地向用户发布内容。通常只在讨论性能问题时会提到它的性能，但它还能节省成本。在优化性能时，向特定用户发布内容的服务器的选择基于对网络可用度的测量。例如，CDN 可能选择网络阶跃数最小的服务器，或者具有最短响应时间的服务器。

一些大型 Internet 公司都拥有他们自己的 CDN，但使用一个 CDN 服务提供商更为有效。Akamai Technologies, Inc.是业界的领头羊。2005 年，Akamai 收购了 Speedera Networks——最早的低成本选择。Mirror Image Internet, Inc.现在和 Akamai 同处领军地位。Limelight Network, Inc.是另外一家竞争者。其他提供商，如 SAVVIS Inc.则专攻利基市场（Niche Market），如视频内容发布。

表 2-1 展示了十大美国 Internet 网站和他们所使用的 CDN 服务提供商。

表 2-1：十大网站使用的 CDN 服务提供商

网站	CDN
http://www.amazon.com	Akamai
http://www.aol.com	Akamai
http://www.cnn.com	
http://www.ebay.com	Akamai、Mirror Image
http://www.google.com	
http://www.msn.com	SAVVIS
http://www.myspace.com	Akamai、Limelight
http://www.wikipedia.org	
http://www.yahoo.com	Akamai
http://www.youtube.com	

可以看到：

- 五家使用 Akamai
- 一家使用 Mirror Image
- 一家使用 Limelight
- 一家使用 SAVVIS
- 四家或不使用 CDN，或者使用自家的 CDN 解决方案

www.TopSage.com

小型的非商业网站可能无法支付这些 CDN 服务的开销。不过有很多免费的 CDN 服务可以使用。Globule（*http://www.globule.org*）是由位于阿姆斯特丹的自由大学（Vrije Universiteit）开发的 Apache 模块。CoDeeN（*http://codeen.cs.princeton.edu*）是普林斯顿大学基于 PlanetLab 构建的。CoralCDN（*http://www.coralcdn.org*）是由纽约大学完成的。它们以不同的方式部署。有些需要最终用户使用一个代理来配置他们的浏览器，有的需要开发者使用不同的域名修改他们的组件的 URL。无论如何也不要使用 HTTP 重定向来将用户指向到本地服务器，这会使 Web 页面反应速度变慢（参见第 11 章）。

除了缩短响应时间之外，CDN 还可以带来其他优势。他们的服务包括备份、扩展存储能力和进行缓存。CDN 还有助于缓和 Web 流量峰值压力，如在获取天气或股市新闻、浏览流行的体育或娱乐事件时。

依赖 CDN 的一个缺点是你的响应时间可能会受到其他网站——甚至很可能是你的竞争对手流量的影响。CDN 服务提供商在其所有客户之间共享其 Web 服务器组。另一个缺点是无法直接控制组件服务器所带来的特殊麻烦。例如，修改 HTTP 响应头必须通过服务提供商来完成，而不是由你的工作团队完成。最后，如果 CDN 服务的性能下降了，你的工作质量也随之下降。从表 2-1 中可以看到，eBay 和 MySpace 都使用了两个 CDN 服务提供商，如果你想左右逢源，这是一种聪明的做法。

CDN 用于发布静态内容，如图片、脚本、样式表和 Flash。提供动态 HTML 页面会引入特殊的存储需求——数据库连接、状态管理、验证、硬件和 OS 优化等。这些复杂性超越了 CDN 的能力范围。另一方面，静态文件更容易存储并具有较少的依赖性。这就是为什么对于地理上分散的用户人群来说，CDN 能轻易地得到响应速度上的提高。

节省
The Savings

这一部分讨论的两个在线示例展示了使用 CDN 得到的响应性能上的改善。两个示例包含相同的测试组件——5 个脚本、1 个样式表和 8 个图片。在第一个例子中，这些组件存储在 Akamai Technologies CDN 上。在第二个例子中，它们放置在一台单独的服务器上。

CDN 的示例

http://stevesouders.com/hpws/ex-cdn.php

无 CDN 的示例

http://stevesouders.com/hpws/ex-nocdn.php

www.TopSage.com

将组件存储在 CDN 上的示例比将所有组件放在一台单独的 Web 服务器上的页面加载起来快 18%（1013ms: 1232ms）。我是在加利福尼亚的家中使用 DSL（900Kbps）测试的。你的测试结果取决于你的连接速度和地理位置。那台单独的 Web 服务器放在华盛顿特区附近。你居住得离华盛顿特区越近，在 CDN 示例中看到的响应时间差别越小。

如果你以你自己的响应时间测试衡量使用 CDN 的优势，千万记住你运行测试的地理位置对结果有着重要影响。例如，基于多数网络公司都在他们的办公室附近选择数据中心这一假设，你正在使用的 Web 客户端很可能位于离当前 Web 服务器很近的地方。因此，如果你在你的浏览器中进行测试，在不使用 CDN 的情况下，响应时间通常会更快一些。需要牢记的是很多用户并不位于离你的 Web 服务器很近的地方。要测量切换到 CDN 的真实影响就必须在多个地理位置上测量响应时间。诸如 Keynote System（*http://www.keynote.com*）和 Gomez（*http://www.gomez.com*）这样的服务有助于完成这种测试。

在 Yahoo!，这一因素确实困扰过我们一段时间。在将 Yahoo! Shopping 转移到 Akamai 之前，我们的初步测试是在 Yahoo!总部的一个实验室中进行的，其位置离 Yahoo!数据中心很近。在使用 Akamai 的 CDN 之后，响应时间的改善——与其实验室中的测量结果相比——不足 5%（并不很让人激动）。但我们知道当我们将 CDN 的变化展示给散布在全世界的真实用户后，响应时间的改善将会更好。在我们将变化呈现给最终用户后，Yahoo! Shopping 网站的响应时间整体减少了 20%，而我们做的仅仅是将所有静态组件转移到了 CDN 上。

使用内容发布网络

www.TopSage.com

第 3 章

规则 3——添加 Expires 头

Rule 3: Add an Expires Header

大家网 TopSage.com

在设计 Web 页面的时候，首次访问的响应时间并不是唯一需要考虑的。如果是这样的话，我们可以将规则 1 发挥到极致，并且不在页面上放置任何图片、脚本和样式表。然而，我们都知道，图片、脚本和样式表能够加强用户体验，尽管这意味着页面需要花更长的时间进行加载。这一章介绍的规则 3 展示了如何配置组件，使其能够最大化地利用浏览器的缓存能力来改善页面的性能。

今天的 Web 页面都包含了大量的组件，并且其数量在不断增长。页面的初访者会进行很多 HTTP 请求，但通过使用一个长久的 `Expires` 头，使这些组件可以被缓存。这会在后续的页面浏览中避免不必要的 HTTP 请求。长久的 `Expires` 头最常用于图片，但应该将其用在所有组件上，包括脚本、样式表和 Flash。很多顶级网站现在都还没有做到这一点。在这一章里，我将指出这些网站并说明为什么他们的页面没有像应该的那样快。添加长久的 `Expires` 头会带来额外的开发成本，如“修订文件名”一节中介绍的那样。

Expires 头

Expires Header

浏览器（和代理）使用缓存来减少 HTTP 请求的数量，并减小 HTTP 响应的大小，使 Web 页面加载得更快。Web 服务器使用 `Expires` 头来告诉 Web 客户端它可以使用一个组件的当前副本，直到指定的时间为止。HTTP 规范中简要地称该头为“在这一日期/时间之后，响应将被认为是无效的”。它在 HTTP 响应中发送。

```
Expires: Thu, 15 Apr 2010 20:00:00 GMT
```


www.TopSage.com

这是一个有效期非常长久的 Expires 头（*far future Expires header*），它告诉浏览器该响应的有效性持续到 2010 年 4 月 15 日为止。如果为页面中的一个图片返回了这个头，浏览器在后续的页面浏览中会使用缓存的图片，将 HTTP 请求的数量减少一个。请参见绪言 B，复习一下 Expires 头和 HTTP。

大家网 TopSage.com

Max-Age 和 mod_expires

Max-Age and mod_expires

在解释缓存如何很好地改善传输性能之前，需要提及除了 Expires 头之外的另一种选择。HTTP 1.1 引入了 Cache-Control 头来克服 Expires 头的限制。因为 Expires 头使用一个特定的时间，它要求服务器和客户端的时钟严格同步。另外，过期日期需要经常检查，并且一旦未来这一天到来了，还需要在服务器配置中提供一个新的日期。

换一种方式，Cache-Control 使用 max-age 指令指定组件被缓存多久。它以秒为单位定义了一个更新窗。如果从组件被请求开始过去的秒数少于 max-age，浏览器就使用缓存的版本，这就避免了额外的 HTTP 请求。一个长久的 max-age 头可以将刷新窗设置为未来 10 年。

```
Cache-Control: max-age=315360000
```

使用带有 max-age 的 Cache-Control 可以消除 Expires 的限制，但对于不支持 HTTP 1.1 的浏览器（尽管这只占你的访问量的 1%以内），你可能仍然希望提供 Expires 头，你可以同时指定这两个响应头——Expires 和 Cache-Control max-age。如果两者同时出现，HTTP 规范规定 max-age 指令将重写 Expires 头。然而，如果你很尽职尽责，你仍然需要担心 Expires 带来的时钟同步和配置维护问题。

幸运的是，mod_expires Apache 模块（*http://httpd.apache.org/docs/2.0/mod/mod_expires.html*）使你在使用 Expires 头时能够像 max-age 那样以相对的方式设置日期。这通过 ExpiresDefault 指令来完成。在下面的例子里，图片、脚本和样式表的过期时间被设计为自请求开始的 10 年之后：

```
<FilesMatch "\.(gif|jpg|js|css)$">
ExpiresDefault "access plus 10 years"
</FilesMatch>
```

时间可以以年、月、周、日、小时、分钟、秒为单位来设置。它同时向响应中发送 Expires 头和 Cache-Control max-age 头。

www.TopSage.com

```
Expires: Sun, 16 Oct 2016 05:43:02 GMT
Cache-Control: max-age=315360000
```

大家网 TopSage.com

实际的过期日期根据何时接到请求而变，但是即便如此，它永远都是 10 年以后。由于 `Cache-Control` 具有优先权并且明确指出了相对于请求时间所经过的秒数，时钟同步问题就被避免了。不用担心固定日期的更新，他在 HTTP 1.0 浏览器中也能够使用。跨浏览器改善缓存的最佳解决方案就是使用由 `ExpiresDefault` 设置的 Expires 头。

对十大网站的调查（参见表 3-1）表明其中的七个网站使用了这种方法，五个网站同时使用了 `Expires` 头和 `Cache-Control max-age` 头。一个网站只是用了 `Expires` 头，还有一个网站只是用了 `Cache-Control max-age` 头。悲哀的是，还有三个网站没使用这种方法。

表 3-1：Expires 和 max-age 的使用情况

网站	Expires	max-age
http://www.amazon.com		
http://www.aol.com	✓	✓
http://www.cnn.com		
http://www.ebay.com	✓	✓
http://www.google.com	✓	
http://www.msn.com	✓	✓
http://www.myspace.com		✓
http://www.wikipedia.org	✓	✓
http://www.yahoo.com	✓	✓
http://www.youtube.com		

空缓存 VS 完整缓存

Empty Cache vs. Primed Cache

只有在用户已经访问过你的网站之后，长久的 `Expires` 头才会对页面浏览产生影响。当用户第一次访问你的网站时，它不会对 HTTP 请求的数量产生任何影响，此时浏览器的缓存是空的。因此，其性能的改进取决于用户在访问你的页面时是否有完整缓存。你的访问流量几乎都来自于那些有完整缓存的用户。使你的组件可缓存能够改善这些用户的响应时间。

当我说“空缓存”或“完整缓存”时，我指的是与你的页面相关的浏览器缓存的状态。如果你的页面中的组件没有放在缓存中，则缓存为“空”。浏览器的缓存可能包含来自其他网站的组件，但这对你的页面没有帮助。反之，如果你的页面中的可缓存组件都在缓存中，则缓存是“完整的”。

www.TopSage.com

空缓存或完整缓存页面浏览的数量取决于 Web 应用程序的本质。一个类似"每日一词"的网站对典型用户来说，每个会话可能只产生一个页面浏览。很多原因会导致当用户下一次访问网站时，"每日一词"的组件可能会不在缓存中：

- 尽管他很渴望掌握更多的词汇，他仍然可能只是每周或者每月访问该页面一次，而不是每天一次。
- 在最近一次访问之后，用户可能手动清空了他的缓存。
- 用户可能访问了太多其他的网站以致缓存已满，"每日一词"的组件被移出缓存。
- 浏览器或杀毒软件可能会在关闭浏览器时清空缓存。

由于每次会话只产生一个页面浏览，"每日一词"的组件不大可能会被缓存，因此带有完整缓存的页面浏览所占的百分比是很低的。

另一方面，在一个旅游网站或 Email 网站中可能每个用户会话能产生多次页面浏览，完整缓存的页面浏览数量就会多一些。在这种情况下，很多页面会在浏览器缓存中发现你的组件。

我们在 Yahoo!对此进行了测量，发现**拥有完整缓存的、每天至少访问一次的唯一用户数占** *40%~60%*，取决于 Yahoo!的内容。同样的研究表明**带有完整缓存的页面浏览数量为** *75%~85%*（注 1）。注意第一个统计测量的是"唯一用户"，而第二个测量的是"页面浏览"。拥有完整缓存的页面浏览所占的百分比要高于拥有完整缓存的唯一用户，因为很多 Yahoo!功能在每个会话中会接收到多次页面浏览。用户在一天中只会有一次使用空缓存，然后很多后续页面访问都将拥有完整缓存。

这些浏览器缓存统计数据解释了为什么优化完整缓存体验是那么的重要。我们希望 40%~60%的用户和 75%~85%的页面浏览的完整缓存能得到优化。这一百分比对你的网站来说有可能不同，但只要用户通常每个月至少访问你的网站一次，或每会话能产生多次页面浏览，这一统计值就会差不多。通过使用长久的 `Expires` 头可以增加被浏览器缓存的组件的数量，并在后续页面浏览中重用它们，而无需通过用户的 Internet 连接发送一个字节。

不仅仅是图片

More Than Just Images

为图片使用长久的 `Expires` 头非常之普遍，但这一最佳实践不应该仅限于图片。长久的 `Expires` 头应该包含任何不经常变化的组件，包括脚本、样式表和 Flash 组件。但是，

注 1：Tenni Theurer，"Performance Research, Part2: Browser Cache Usage - Exposed!"，*http://yuiblog.com/blog/2007/01/04/performance-research-part-2*。

www.TopSage.com

HTML 文档不应该使用长久的 Expires 头，因为它包含动态内容，这些内容在每次用户请求时都将被更新。

理想情况下，页面中的所有组件都应该具有长久的 Expires 头，并且后续的页面浏览中只需为 HTML 文档进行一个 HTTP 请求。当文档中的所有组件都是从浏览器缓存中读取出来时，响应时间会减少 50%或更多。

我调查了美国十大 Internet 网站，并记录了有多少图片、脚本和样式表使用了 Expires 或 Cache-Control max-age 头并设置了至少 30 天以上。如表 3-2 所示，但这看起来并不好。这里统计了三类组件——图片、样式表和脚本。表 3-2 展示了可缓存 30 天以上的组件和对应的每种类型组件的总数。我们来看一下这些网站在实践缓存组件时采用的时间范围：

- 5 个网站使其大部分**图片**可缓存 30 天以上。
- 4 个网站使其大部分**样式表**可缓存 30 天以上。
- 2 个网站使其大部分**脚本**可缓存 30 天以上。

表 3-2：带有 Expires 头的组件

网站	图片	样式表	脚本	Last-Modified Δ 中值
http://www.amazon.com	0/62	0/1	0/3	114 天
http://www.aol.com	23/43	1/1	6/18	217 天
http://www.cnn.com	0/138	0/2	2/11	227 天
http://www.ebay.com	16/20	0/2	0/7	140 天
http://froogle.google.com	1/23	0/1	0/1	454 天
http://www.msn.com	32/35	1/1	3/9	34 天
http://www.myspace.com	0/18	0/2	0/2	1 天
http://www.wikipedia.org	6/8	1/1	2/3	1 天
http://www.yahoo.com	23/23	1/1	4/4	-
http://www.youtube.com	0/32	0/3	0/7	26 天

表 3-2 中的整体百分比指出，所有组件中 74.7%或者是不可缓存的，或者可缓存但时间不超过 30 天。一种可能的解释是，这些组件是不应该缓存的。例如像 *cnn.com* 这样的新闻网站，其 138 个图片 0 个可缓存，可能是有太多的新闻图片，由于需要更新，这些图片会经常刷新，而不是缓存在用户的浏览器中。如果组件是因为经常变化而不被缓存，我们期望看到很近的 Last-Modified 日期。

www.TopSage.com

表 3-2 展示了所有未缓存组件的 Last-Modified 增量（当前日期和 Last-Modified 日期之间的差）中值。在这里 *cnn.com* 的 Last-Modified 增量中值是 227 天。一半的未缓存组件 227 天以来都没有更改过，因此图片的刷新并不是问题所在。

这也是 Yahoo!过去的情况。过去，Yahoo!没有任何可缓存的脚本、样式表和图片。不缓存这些组件背后的逻辑是，用户应该每次都请求它们以便获得更新，因为它们经常改变。然而，在发现实际上这些文件改变得不那么频繁之后，我们意识到将它们缓存起来可以得到更好的用户体验。尽管存在着额外的开发成本，Yahoo!选择将它们做成可缓存的，这将在下一部分介绍。

修订文件名
Revving Filenames

如果我们将组件配置为可以由浏览器代理缓存，当这些组件改变时用户如何获得更新呢？当出现了 `Expires` 头时，直到过期日期为止一直会使用缓存的版本。浏览器不会检查任何更新，直到过了过期日期。这也是为什么使用 `Expires` 头能够显著地减少响应时间——浏览器直接从硬盘上读取组件而无需生成任何 HTTP 流量。因此，即使在服务器上更新了组件，已经访问过网站的用户也不大可能获取最新的组件（因为前一个版本已经在他们的缓存中了）。

为了确保用户能获取组件的最新版本，需要在所有 HTML 页面中修改组件的文件名。Mark Nottingham 的 Web 文章“Caching Tutorial for Web Authors and Webmasters”称：

> 最有效的解决方案是修改其所有链接，这样，全新的请求将从原始服务器下载最新的内容。

取决于你如何构造 HTML 页面，这项工作可能很轻松也可能很痛苦。如果你使用 PHP、Perl 等动态语言生成 HTML 页，一种简单的解决方案就是为所有组件的文件名使用变量。使用这种方法，在页面中更新文件名只需要简单地在某个地方修改变量。在 Yahoo!我们经常将这一步作为生成过程的一部分——将版本号嵌在组件的文件名中（例如 *yahoo_2.0.6.js*），而且在全局映射中修订过的文件名会自动更新。嵌入版本号不仅可以改变文件名，还能在调试时更容易地找到准确的源代码文件。

www.TopSage.com

大家网 TopSage.com

示例
Examples

下面两个示例演示了使用长久的 Expires 头能够得到的性能改进。这两个示例包含相同的组件——六个图片、三个脚本和一个样式表。在第一个示例中，这些组件**没有**使用长久的 Expires 头，而第二个示例使用了。

无 Expires 的示例

http://stevesouders.com/hpws/expiresoff.php

长久的 Expires 的示例

http://stevesouders.com/hpws/expireson.php

添加长久的 Expires 头可以将后续页面浏览的响应时间从 600 毫秒降低到 260 毫秒，这是在 900Kbps 的 DSL 上测试的，减少了 *57%*。页面中的组件越多，响应时间改善得越多。如果你的页面平均超过 6 个图片、3 个脚本和 1 个样式表，页面的速度提升就会超过这个例子中的 57%。

节省下来的这些时间究竟是从哪里来的呢？我在前面曾提到过，一个具有长久 Expires 头的组件将会被缓存，在后续请求时浏览器直接从硬盘上读取它，避免了一个 HTTP 请求。然而，我并没有介绍相反的情况。如果一个组件**没有**长久的 Expires 头，它仍然会存储在浏览器的缓存中。在后续请求中，浏览器会检查缓存并发现组件已经过期（HTTP 术语称之为"陈旧"）。为了提高效率，浏览器会向原始服务器发送一个**条件 GET 请求**（*Conditional Get Request*）。请参见绪言 B 中的示例。如果组件没有改变，原始服务器可以免于发送整个组件，而是发送一个很小的头，告诉浏览器可以使用其缓存的组件。

这些条件请求加起来，就是节省的时间。很多时候，正如我们在十大网站中看到的那样，组件并没有更改，而浏览器总是从磁盘上读取它们。通过使用 Expires 头来避免额外的 HTTP 请求，可以减少一半的响应时间。

为组件添加长久的 Expires 头

www.TopSage.com

第 4 章

规则 4——压缩组件

Rule 4: Gzip Components

前端工程师作出的决定可以显著地减少在网络上传送 HTTP 请求和响应所花的时间。的确，用户的带宽速度、Internet 服务提供商、对等交换点的距离和其他因素超出了开发团队的控制范围。然而，仍然有很多变数可以影响响应时间。规则 1 和规则 3 通过限制不必要的 HTTP 请求解决了响应时间的问题。如果没有 HTTP 请求——理想情况下——也就没有了网络活动。规则 2 通过将 HTTP 响应拉近用户来减少响应时间。

本章将要介绍的规则 4 通过减小 HTTP 响应的大小来减少响应时间。如果 HTTP 请求产生的响应包很小，传输时间就会减少，因为只需要将很小的包从服务器传递到客户端。这一效果对速度较慢的带宽尤其明显。本章展示了如何使用 gzip 编码来压缩 HTTP 响应包，并由此减少网络响应时间。这是减小页面大小的最简单的技术，但影响是最大的。还有很多方式可以减小 HTML 文档的页面大小（删除注释和缩短 URL 等），但它们需要更多的工作，且收效甚微。

压缩是如何工作的

How Cpmpression Works

用于减小文件体积的文件压缩已经在 Email 应用和 FTP 站点中使用了 10 年，同样的技术也可以用于向浏览器发布压缩的 Web 页面。从 HTTP 1.1 开始，Web 客户端可以通过 HTTP 请求中的 `Accept-Encoding` 头来标识对压缩的支持。

```
Accept-Encoding: gzip, deflate
```


www.TopSage.com

大家网
TopSage.com

如果 Web 服务器看到请求中有这个头，就会使用客户端列出来的方法中的一种来压缩响应。Web 服务器通过响应中的 `Content-Encoding` 头来通知 Web 客户端。

```
Content-Encoding: gzip
```

gzip 是目前最流行和最有效的压缩方法。这是 GNU 项目开发的一种免费的格式（也就是说在专利权和其他限制方面没有任何阻碍）并被标准化为 RFC 1952。你可能见到的另一种压缩格式是 *deflate*，但其效果略逊且不太流行。事实上，我只见过一个使用 deflate 的网站——*msn.com*。支持 deflate 的浏览器也支持 gzip，但很多浏览器支持 gzip 却不支持 deflate，因此 gzip 是最理想的压缩方法。

压缩什么
What to Compress

服务器基于文件类型选择压缩什么，但这通常受限于对其进行的配置。很多网站会压缩其 HTML 文档。压缩脚本和样式表也是非常值得的，但很多网站没有这样做（实际上，值得压缩的内容包括 XML 和 JSON 在内的任何文本响应，但这里只关注脚本和样式表，因为它们用得最普遍）。图片和 PDF **不**应该压缩，因为它们本来就已经被压缩了。试图对它们进行压缩只会浪费 CPU 资源，还有可能会增加文件大小。

压缩的成本有——服务器端会花费额外的 CPU 周期来完成压缩，客户端要对压缩文件进行解压缩。要检测收益是否大于开销，需要考虑响应的大小、连接的带宽和客户端与服务器之间的 Internet 距离。这些信息通常难以得到，即便得到了，也有很多其他变数需要考虑。根据经验通常对大于 1KB 或 2KB 的文件进行压缩。`mod_gzip_minimum_file_size` 指令控制着希望压缩的文件的最小值，默认值是 500B。

我观察了美国 10 个最流行的网站的 gzip 使用情况。9 个网站压缩了其 HTML 文档，7 个网站压缩了**大多数**脚本和样式表，只有 5 个网站压缩了**所有的**脚本和样式表。没有压缩任何 HTML 文档、样式表和脚本的网站错过了将其页面大小减少 70%以上的机会，这将在下一节“节省”中介绍。表 4-1 列出了主流网站及其对压缩的选择。

www.TopSage.com

大家网 TopSage.com

表 4-1：美国十大流行网站的 gzip 使用情况

网站	gzip HTML	gzip 脚本	gzip 样式表
http://www.amazon.com	✓		
http://www.aol.com	✓	一些	一些
http://www.cnn.com			
http://www.ebay.com	✓		
http://froogle.google.com	✓	✓	✓
http://www.msn.com	✓	deflate	deflate
http://www.myspace.com	✓	✓	✓
http://www.wikipedia.org	✓	✓	✓
http://www.yahoo.com	✓	✓	✓
http://www.youtube.com	✓	一些	一些

节省
The Savings

压缩通常能将响应的数据量减少将近 70%。表 4-2 列出了脚本和样式表（有大有小）的压缩示例。除了 gzip 外，使用 deflate 的结果也显示了出来。

表 4-2：使用 gzip 和 deflate 的压缩大小

文件类型	未压缩大小	gzip 大小	gzip 节省	deflate 大小	deflate 节省
脚本	3 277 字节	1 076 字节	67%	1 112 字节	66%
脚本	39 713 字节	14 488 字节	64%	16 583 字节	58%
样式表	968 字节	426 字节	56%	463 字节	52%
样式表	14 122 字节	3 748 字节	73%	4 665 字节	67%

从表 4-2 可以清楚地看到 gzip 是典型的压缩选择。gzip 能将响应整体减小 66%，而 deflate 能减小 60%。对于这些文件，gzip 能比 deflate 多压缩 6%。

配置
Configuration

配置 gzip 时使用的模块取决于 Apache 的版本——Apache 1.3 使用 `mod_gzip` 而 Apache 2.x 使用 `mod_deflate`。这一节介绍如何配置每个模块，但仅介绍 Apache，因为它是 Internet 上最流行的 Web 服务器。

www.TopSage.com

Apache 1.3——mod_gzip

Apache 1.3 的 gzip 压缩由 mod_gzip 模块提供。有很多 mod_gzip 配置指令，在 mod_gzip 的网站（*http://www.schroepl.net/projekte/mod_gzip*）上有所描述。以下是最常用的指令：

mod_gzip_on

启用 mod_gzip

mod_gzip_item_include
mod_gzip_item_exclude

基于文件类型、MIME 类型、用户代理等定义哪些需要压缩、哪些不需要。

很多 Web 主机服务都默认为 text/html 打开了 mod_gzip。你要做的最重要的配置修改就是需要明确压缩脚本和样式表。可以使用下面的 Apache 1.3 指令来完成：

```
mod_gzip_item_include         file      \.js$
mod_gzip_item_include         mime     ^application/x-javascript$
mod_gzip_item_include         file      \.css$
mod_gzip_item_include         mime     ^text/css$
```

gzip 命令行工具提供了一个选项，用于控制压缩的程度，可以在 CPU 使用量和数据大小的变化之间进行取舍，但 mod_gzip 中没有配置指令能够控制压缩级别。如果流式压缩产生的 CPU 负载成问题，可以考虑在磁盘或内存中缓存经过压缩的组件。手工压缩响应和更新缓存增加了你的维护工作，并可能成为一种负担。幸运的是，mod_gzip 提供了选项，可以将保存压缩过的内容自动保存在磁盘上，并在原内容发生变化时更新压缩过的内容。使用 mod_gzip_can_negotiate 和 mod_gzip_update_static 指令可以完成这一任务。

Apache 2.x——mod_deflate

Apache 2.x 中的压缩通过 mod_deflate 模块来完成。尽管该模块的名字是这样的，但它使用 gzip 来进行压缩。上一节中对压缩脚本和样式表进行的基本配置可以在一行中完成：

```
AddOutputFilterByType DEFLATE text/html text/css application/x-javascript
```

和 mod_gzip 不同，mod_deflate 包含了一个用于控制压缩级别的指令——DeflateCompressionLevel。关于更多配置的内容，请参见 Apache 2.0 mod_deflate 的文档——*http://httpd.apache.org/docs/2.0/mod/mod_deflate.html*。

www.TopSage.com

大家网
TopSage.com

代理缓存

Proxy Caching

当浏览器直接与服务器通信时，迄今为止所介绍的配置都能很好地工作。Web 服务器基于 Accept-Encoding 来检测是否对响应进行压缩。不管是否压缩过，浏览器都会基于响应中的其他 HTTP 头如 Expires 和 Cache-Control（参见第 3 章）来缓存响应。

当浏览器通过代理来发送请求时，情况就变得复杂了。假设针对某个 URL 发送到代理的第一个请求来自于一个不支持 gzip 的浏览器。这是到达代理的第一个请求，因此其缓存为空。代理会将请求转发到 Web 服务器。此时服务器的响应是未经过压缩的。这个没有压缩的响应被代理缓存起来并发送给浏览器。现在，假设到达代理的第二个请求访问的是同一个 URL，来自于一个支持 gzip 的浏览器。代理会使用其缓存中（未经压缩）的内容进行响应，这就失去了进行压缩的机会。如果顺序反了——第一个请求来自于一个支持 gzip 的浏览器，而第二个请求来自于一个不支持 gzip 的浏览器——情况可能更严重。在这种情况下，代理的缓存中拥有内容的一个压缩版本，并将这个版本提供给后续的浏览器，而不管它们是否支持 gzip。

解决这一问题的方法是在 Web 服务器的响应中添加 Vary 头。Web 服务器可以告诉代理根据一个或多个请求头来改变缓存的响应。由于压缩的决定是基于 Accept-Encoding 请求头的，因此需要在服务器的 Vary 响应头中包含 Accept-Encoding。

```
Vary: Accept-Encoding
```

这将使得代理缓存响应的多个版本，为 Accept-Encoding 请求头的每个值缓存一份。在前面的例子中，代理会缓存每个响应的两个版本——Accept-Encoding 为 gzip 时的压缩内容和没有指定 Accept-Encoding 时的非压缩内容。当浏览器带着 Accept-Encoding: gzip 访问代理时，它接收到的是压缩过的内容。没有 Accept-Encoding 请求头的浏览器收到的是未经压缩的内容。默认情况下，mod_gzip 会向所有响应添加 Vary: Accept Encoding 头，以驱使代理执行正确的操作。与 Vary 有关的更多信息，请访问 *http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.44*。

www.TopSage.com

大家网 TopSage.com

边缘情形
Edge Cases

服务器和客户端的压缩对等性看似简单，但必须正确才行。无论是客户端还是服务器发生错误（发送压缩内容到不支持它的客户端、忘记将压缩内容声明为已经进行了 gzip 编码等），页面都会被破坏。错误并不会经常发生，但它们是必须考虑的**边缘情形**（*Edge Case*）。

今天大约 90%的通过浏览器进行的 Internet 通信都需要使用 gzip。如果一个浏览器宣称支持 gzip，你通常可以相信它。但 Internet Explorer 的早期未打补丁的版本存在着一些已知的缺陷，尤其是 Internet Explorer 5.5 和 Internet Explorer 6.0 SP1，Microsoft 发布了两篇 Knowledge Base 文章（*http://support.microsoft.com/kb/313712/en-us* 和 *http://support.microsoft.com/kb/312496/en-us*）用来说明此问题。还有其他一些问题，但这些问题只发生在不到 Internet 通信量 1%的浏览器上。一种安全的方式是只为已经证实过支持压缩的浏览器提供压缩内容，如 Internet Explorer 6.0 及之后版本和 Mozilla 5.0 及之后版本。这被称作**浏览器白名单**（*Browser Whitelist*）方式。

使用这种方式，可能会失去为一小部分本来支持压缩的浏览器提供压缩内容的机会。但其他选择——为不支持压缩的的浏览器提供压缩内容——更为糟糕。在 Apache 1.3 的 `mod_gzip` 中，可以通过在 `mod_gzip_item_include` 中使用恰当的 `User-Agent` 值来指定浏览器白名单：

```
mod_gzip_item_include reqheader "User-Agent: MSIE [6-9]"
mod_gzip_item_include reqheader "User-Agent: Mozilla/[5-9]"
```

在 Apache 2.x 中可以使用 BrowserMatch 指令：

```
BrowserMatch ^MSIE [6-9] gzip
BrowserMatch ^Mozilla/[5-9] gzip
```

把代理缓存加进来后，处理这些边缘情形浏览器将变得更为复杂。你不可能和代理共享浏览器白名单配置。用于设置浏览器白名单的指令过于复杂，无法使用 HTTP 头进行编码。最佳做法是将 `User-Agent` 作为代理的另外一种评判标准添加到 `Vary` 头中。

```
Vary: Accept-Encoding,User-Agent
```

当 `mod_gzip` 检测到你在使用浏览器白名单时，它会自动将 `User-Agent` 字段添加到 `Vary` 头。不幸的是，`User-Agent` 有上千种不同的值。代理不太可能为其所代理的所有 URL 缓存 `Accept-Encoding` 和 `User-Agent` 的全部组合。`mod_gzip` 文档（*http://www.schroepl.*

www.TopSage.com

net/projekte/mod_gzip/cache.htm）甚至说："使用对 User-Agent HTTP 头进行求值的过滤器规则将会导致完全禁用为响应包进行的缓存。"因为这实际上破坏了代理缓存。另外一种方式是使用 `Vary: *`或 `Cache-Control: private` 头来禁用代理缓存。因为 `Vary: *`头防止了浏览器使用缓存的组件，最好使用 `Cache-Control: private`，Google 和 Yahoo! 都使用了这种方式。记住这是为所有浏览器禁用代理缓存，因此会增加你的带宽开销，因为代理无法缓存你的内容。

如何平衡压缩和代理支持的决定是很复杂的，需要在加快响应时间、减小带宽开销和边缘情形浏览器缺陷之间进行权衡。正确的答案取决于你的网站——

- 如果你的网站用户很少，并且他们处在一个小圈子中（例如，他们在一个 intranet 中，或者都使用 Firefox 1.5），边缘情形浏览器就不需要太多关注。可以压缩内容并使用 `Vary: Accept-Encoding`。这样可以通过减小组件的大小和利用代理缓存来改善用户体验。
- 如果你更注意带宽开销，可以和前一种情况一样——压缩内容并使用 `Vary: Accept-Encoding`。这降低了服务器端的带宽开销并提升了代理处理的请求数量。
- 如果你拥有大量的、多变的用户群，能够应付较高的带宽开销，并且享有高质量的名声，请压缩内容并使用 `Cache-Control: Private`。这禁用了代理但避免了边缘情形缺陷。

还有一种代理的边缘情形需要指出。这个问题是，默认情况下，ETag（第 13 章中介绍）不能反映出内容是否被压缩，因此代理可能会向浏览器提供错误的内容。此问题在 Apache 的缺陷数据库（*http://issues.apache.org/bugzilla/show_bug.cgi?id=39727*）中有所描述。最好的解决办法是禁用 ETag。由于这是第 13 章中提出的解决方案，我将在那里再对其进行详细介绍。

压缩的实际效果
Gzip in Action

规则 4 的三个示例展示了你可以在你的网站中部署的几种不同程度的压缩。

无压缩的示例

http://stevesouders.com/hpws/nogzip.html

压缩 HTML 的示例

http://stevesouders.com/hpws/gzip-html.html

压缩所有组件的示例

http://stevesouders.com/hpws/gzip-all.html

www.TopSage.com

除了48.6KB的HTML文档之外，每个示例页面还包含一个59.9KB的样式表和一个68.0KB的脚本。表4-3展示了总页面大小是如何随着执行的压缩情况而变化的。经过压缩HTML文档、样式表和脚本，页面的大小从177.6KB减少到46.4KB，减小了*73.8%*！压缩通常能将内容压缩约70%，但这会随着空白及重复字符的数量而变。

表4-3：不同程度压缩节省的页面大小

示例	组件（HTML、CSS、JS）	总大小	节省的大小	响应时间	节省的时间
不进行任何压缩	48.6KB、59.9KB、68.0KB	177.6KB	-	1562ms	-
压缩HTML	13.9KB、59.9KB、68.0KB	141.9KB	34.7KB（19.7%）	1411ms	151ms（9.7%）
压缩所有组件	13.9KB、14.4KB、18.0KB	46.4KB	130.2KB（73.8%）	731ms	831ms（53.2%）

压缩了所有组件的页面加载起来比未进行压缩的示例运行时间减少了831毫秒，响应时间减少了*53.2%*。这是在900Kbps DSL线路上测量的。绝对响应时间可能会发生变化，这取决于Internet连接、CPU、浏览器、地理位置，等等。然而，相对的节省仍是差不多的。对Web服务器配置进行简单的修改，压缩尽可能多的组件，就能显著改善页面的反应速度。

压缩脚本和样式表

www.TopSage.com

规则 5——将样式表放在顶部
Rule 5: Put Stylesheets at the Top

Yahoo!的一个负责主要门户的团队向他们的页面中添加了大量 DHTML 特性，并尝试确保它们不会对响应时间产生负面影响。其中一个复杂的 DHTML 特征是在发送 Email 消息时弹出一个 DIV，这不属于实际呈现页面的一部分——它只在页面加载完毕后，用户单击按钮来发送 Email 消息时才会访问。由于在呈现页面时不需要使用它，前端工程师将弹出式 DIV 所需的 CSS 放在了外部样式表中，并在页面的底部添加了相应的 LINK 标签，期望将其包含在页面的末尾能够使其加载更快。

这背后的逻辑是有意义的。其他很多组件（图片、样式表、脚本，等等）是呈现页面所必需的。由于组件（通常）是按照它们在文档中出现的顺序下载的，将有 DHTML 特性的样式表放在最后可以使得很多重要的组件首先被下载，从而得到一个加载很快的页面。

果真如此吗？

在 Internet Explorer（仍然是最流行的浏览器）中，实际产生的页面比原来的明显缓慢。在尝试寻找一种能为页面加速的方法时，我们发现将 DHTML 特征的样式表放在文档顶部——Head 中——能使页面加载得更快。这与我们所期望的相矛盾。将样式表放在前面，会延迟页面中其他重要组件的加载，怎么会改善页面加载时间呢？对此更深入的研究导致了规则 5 的出现。

逐步呈现
Progressive Rendering

关心性能的前端工程师都希望页面能逐步地加载，也就是说，我们希望浏览器能够尽快显示内容。这对于有很多内容的页面以及 Internet 连接很慢的用户来说尤其重要。为用户提

www.TopSage.com

供可视化回馈的重要性已经有了很好的研究和记载。Jakob Nielson——可用性工程师先驱——在其 Web 文章（注 1）中从进度指示器的角度强调了可视化回馈的重要性。

> 进度指示器有三个主要优势——它们让用户知道系统没有崩溃，只是正在为他或她解决问题；它们指出了用户大概还需要等多久，以便用户能够在漫长的等待中做些其他事情；最后，它们能给用户提供一些可以看的东西，使得等待不再是那么无聊。最后一点优势不可低估，这也是为什么推荐使用图形进度条而不是仅仅以数字形式显示预期的剩余时间。

在我们这里，HTML 页面就是进度指示器。当浏览器逐步地加载页面时，页头、导航栏、顶端 logo 等，所有这些都会为等待页面的用户提供视觉反馈。这改善了整体用户体验。

将样式表放在文档底部会导致在浏览器中阻止内容逐步呈现。为避免当样式变化时重绘页面中的元素，浏览器会阻塞内容逐步呈现。规则 5 对于加载页面所需的实际时间没有太多影响，它影响更多的是浏览器对这些组件顺序的反应。实际上，用户**感觉缓慢**的页面反而是可视化组建加载得**更快**的页面。在浏览器和用户等待位于底部的样式表时，浏览器会延迟显示任何可视化组件。下一节给出的示例展示了这一现象，我将其称之“白屏”。

sleep.cgi

在为这一现象创建示例时，我开发了一个工具，它在展示延迟的组件如何影响 Web 页面上十分有效——*sleep.cgi*。这是一个简单的 Perl CGI 程序，它需要下面这些参数——

`sleep`

: 将响应延迟多长时间（按秒计）。默认值为 `0`。

`type`

: 返回的组件类型。可取的值只有 `gif`、`js`、`css`、`html` 和 `swf`。默认值为 `gif`。

`expires`

: 下面三个值之一：`-1`（返回一个已经过时的 `Expires` 头）、`0`（不返回 `Expires` 头）和 `1`（返回一个未来的 `Expires` 头）。默认值为 `1`。

注 1：Jakob Nielson，"Response Times: The Three Important Limits"，*http://www.useit.com/papers/responsetime.html*。

www.TopSage.com

大家网 TopSage.com

last

取 −1 返回一个 Last-Modified 头，其时间戳和文件的时间相等。取 0 则不返回 Last-Modified 头。默认值为 −1。

redir

取 1 则产生一个 302 响应，重定向回同样的、不带 redir=1 的 URL。

第一个示例需要一些慢速的图片和一个慢速的样式表。这通过下面得到 *sleep.cgi* 的请求来实现——

```
<img src="/bin/sleep.cgi?type=gif&sleep=2&expires=-1&last=0">
<link rel="stylesheet" href="/bin/sleep.cgi?type=css&sleep=1&expires=-1&last=0">
```

图片和样式表都使用 expires=−1 选项时，会得到一个拥有已过期的 Expires 头的响应。这样就避免了组件被缓存，你可以重复运行这个测试，每次都能得到同样的体验（我还为每个组件的 URL 添加了唯一的时间戳，进一步防止缓存）。为了减少这个测试中的变量，我指定了 last=0 来从响应中移除 Last-Modified 头。图片需要两秒的延迟（sleep=2），而样式表只需要一秒的延迟（sleep=1）。这确保了看到的延迟不是由样式表的响应时间造成的，而是由阻塞行为造成的（这也是该页面要测试的）。

放大组件的响应时间，使得我们可以形象地看到它们在页面加载和响应时间上的效果。我公开了该 Perl 代码，这样其他人也可以使用该代码完成他们的测试（*http://stevesouders.com/hpws/sleep.txt*）。请将代码复制到一个可执行文件中并为其命名为 *sleep.cgi*，然后将其放到 Web 服务器上的一个可执行目录中即可。

白屏
Blank White Screen

这一节展示了两个 Web 页面，它们只在一个方面有所不同——样式表是在页面的顶部还是底部。我们可以看到这给用户体验带来了多么大的差别！

将 CSS 放在底部
CSS at the Bottom

第一个示例展示了将样式表放在 HTML 文档底部所带来的不足。

将 CSS 放在底部的示例

http://stevesouders.com/hpws/css-bottom.php

www.TopSage.com

注意将样式表放在文档底部是如何延迟页面加载的。这个问题很难跟踪到，因为它只发生在 Internet Explorer 中，并且依赖于页面是如何加载的。在使用过这个页面之后，你会发现偶然发生页面加载缓慢。当发生这种现象时，页面会完全空白，直到页面所有的内容同时涌上屏幕，如图 5-1 所示，逐步呈现被禁止了。这是一种不好的用户体验，因为没法向用户确保他的请求正在被正确地处理，用户会因为不知道发生了什么而离开。这就是用户离开你的网站投奔竞争对手的原因。

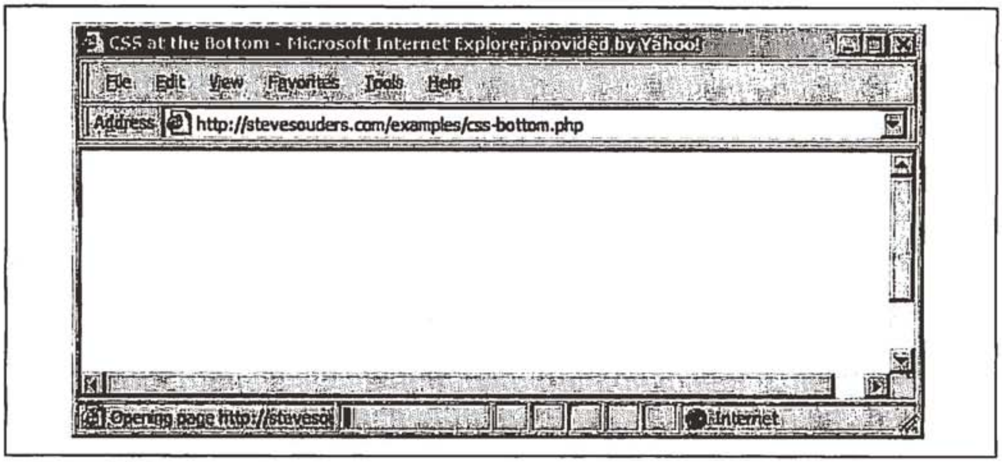


图 5-1：白屏

在 Internet Explorer 中，将样式表放在文档底部会导致白屏问题的情形有以下几种：

在新窗口中打开时

单击示例页面中的“new window”链接，在新窗口中打开“CSS at the Bottom”。用户通常在跨站导航时打开新窗口，如从搜索结果页导航到实际的目标页。

重新加载时

单击刷新按钮是另外一种导致白屏的方式，这是一种常见的用户操作。在页面加载时最小化然后恢复窗口就能看到白屏。

作为主页

将浏览器默认页设置为 *http://stevesouders.com/hpws/css-bottom.php* 并打开新的浏览器窗口就会导致白屏。规则 5 对于那些希望其网站能被用作主页的团队来说尤其重要。

www.TopSage.com

将 CSS 放在顶部

CSS at the Top

为了避免白屏，请将样式表放在文档顶部的 HEAD 中。经过这样修改后的示例网站称作“CSS at the Top”，解决了所有错误情况。不管页面是如何加载的——在新窗口打开、重新加载者作为主页——页面都是逐步呈现的。

将 CSS 放在顶部的示例

http://stevesouders.com/hpws/css-top.php

搞定！只有一点比较复杂的地方需要指出。

将样式表包含在文档中有两种方式：使用 LINK 标签和@import 规则。使用 LINK 标签的示例如下所示：

```
<link rel="stylesheet" href="styles1.css">
```

下面是使用带有@import 规则的 STYLE 标签示例：

```
<style>
@import url("styles2.css");
</style>
```

一个 STYLE 块可以包含多个@import 规则，但@import 规则必须放在所有其他规则之前。我曾遇到过没有注意到这一点的情形，开发人员需要花时间去尝试检查为什么@import 规则中的样式表没有加载。出于这一原因，我更喜欢使用 LINK 标签（需要了解的东西较少）。除了语法更简单外，使用 LINK 标签来代替@import 还能带来性能上的收益。@import 规则有可能会导致白屏现象，即便把@import 规则放在文档的 HEAD 标签中也是如此，如下面的示例所示。

将 CSS 放在顶端并使用@import 的示例

http://stevesouders.com/hpws/css-top-import.php

使用@import 规则会导致组件下载时的无序性。图 5-2 展示了前面三个示例的 HTTP 流量。每个页面包含八个 HTTP 请求：

- 一个 HTML 页面
- 六个图片
- 一个样式表

css-bottom.php 和 *css-top.php* 中的组件是按照它们出现的顺序下载的。然而，尽管 *css-top-import.php* 将样式表放在了文档顶部的 HEAD 中，样式表依然是最后下载的，因为它使用了@import。结果，它产生了白屏问题，就像 *css-bottom.php* 一样。

www.TopSage.com

图 5-2 还表明了每个页面加载的整体时间（包括所有页面组建）是一样的——约 7.3 秒。令人惊讶的是，感觉缓慢的页面——*css-bottom.php* 和 *css-top-import.php*——实际下载页面所必需的组件的时间是最短的。它们完成下载 HTML 页面和所有 6 个图片用了 6.3 秒，而 *css-top.php* 用了 7.3 秒来下载页面所必需的组件。css-top.php 多花了 1 秒钟时间，因为它要首先下载样式表，尽管这并不是页面呈现时所必需的。这就使 6 个图片的下载时间延迟了大约 1 秒。尽管花了更多时间来下载所需的组件，但用户感觉 *css-top.php* 显示得更快，因为页面是逐步呈现的。

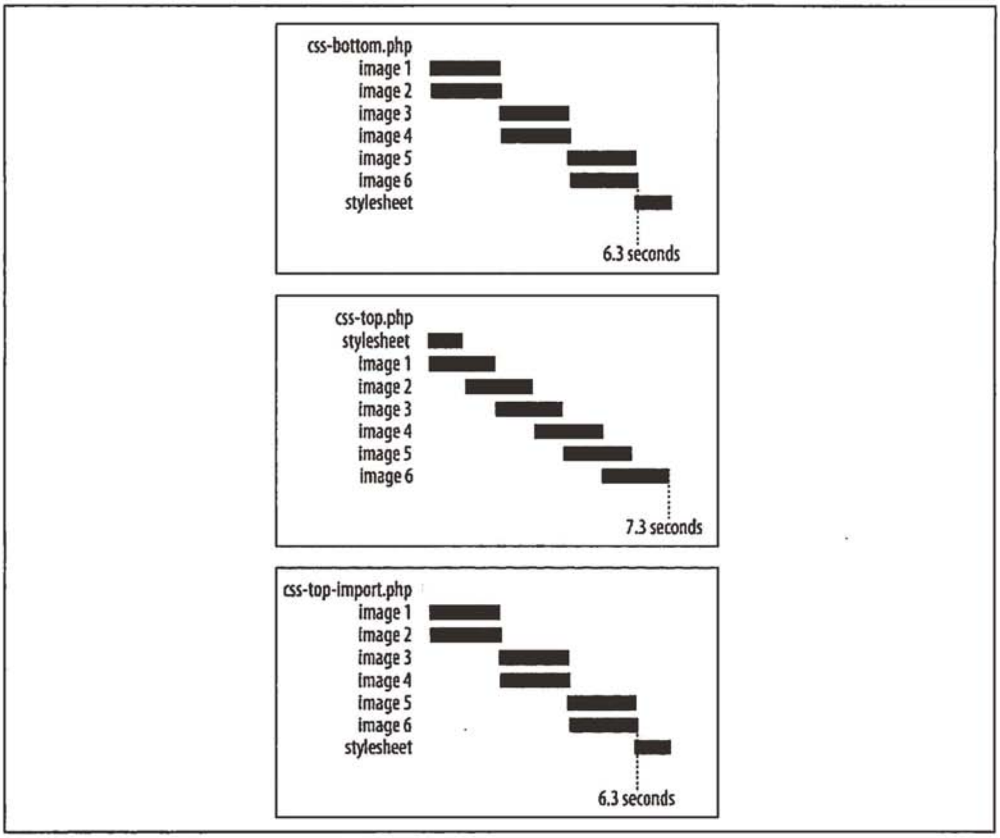


图 5-2：加载组件

太棒了！我们知道怎么做了——**使用** `LINK` **标签将样式表放在文档的** `HEAD` **中**。但如果你像我一样，你就会问自己，“为什么浏览器以这种方式工作呢？”

www.TopSage.com

大家网 TopSage.com

无样式内容的闪烁

Flash of Unstyled Content

白屏现象源自于浏览器的行为。要知道我们的样式表在呈现页面时几乎用不到——它只影响发送 Email 消息时的 DHTML 特性。尽管 Internet Explorer 已经得到了所需的组件，它依然要等到样式表下载完毕之后再呈现它们。样式表在页面中的位置并不影响下载时间，但是会影响页面的呈现。David Hyatt 就为什么浏览器会这样做进行了很好的解释（注 2）。

> 如果样式表仍在加载，构建呈现树就是一种浪费，因为在所有样式表加载并解析完毕之前无需绘制任何东西。否则，在其准备好之前显示内容会遇到 FOUC（无样式内容的闪烁，Flash of Unstyled Content）问题。

下面的例子展示了这一问题。

无样式内容的 CSS 闪烁的示例

http://stevesouders.com/hpws/css-fouc.php

在这个例子中，文档为样式表使用了一个 CSS 规则，但样式表被（不正确地）放在了底部。当页面逐步加载时，文字首先显示，然后是图片。最后，在样式表正确地下载并解析之后，已经呈现的文字和图片要用新的样式重绘了。这就是“无样式内容的闪烁”。应该避免这个问题。

白屏是浏览器在尝试修改前端工程师所犯的错误——将样式表放在文档比较靠后的位置。白屏是对 FOUC 问题的弥补。浏览器可以延迟呈现，直到所有的样式表都下载完之后，这就导致了白屏。反之，浏览器可以逐步呈现，但要承担闪烁的风险。这里没有完美的选择。

前端工程师应该做什么？

What's a Frontend Engineer to Do?

那么如何同时避免白屏和无样式内容的闪烁呢？

在“无样式内容的 CSS 闪烁”示例中，闪烁并不总是发生，它取决于你的浏览器以及如何加载页面。在本章前面的内容中，我曾介绍过白屏仅当在新窗口中加载页面、重新加载和作为主页时在 Internet Explorer 中发生。在这些情况下，Internet Explorer 选择了白屏。然而，如果你单击链接、使用书签或键入 URL，Internet Explorer 选择第二种方式——承担 FOUC 风险。

注 2：David Hyatt，“Surfin' Safari”博客，*http://weblogs.mozillazine.org/hyatt/archives/2004_05.html#005496*。

www.TopSage.com

Firefox 则是一致的——它总是选择第二种方式（FOUC）。所有三个例子在 Firefox 中的行为都是一样的——它们会逐步呈现。对于第一个例子，Firefox 的行为更考虑用户感受，因为样式表对于呈现页面来说并不是必需的，但在“无样式内容的 CSS 闪烁”示例中，用户就不那么幸运了。用户会切实体验到 FOUC 问题，因为 Firefox 是逐步呈现的。

当浏览器的行为不同时，前端工程师应该做些什么呢？

你可以在 HTML 规范中找到答案（*http://www.w3.org/TR/html4/struct/links.html#h-12.3*）：

> 和 A 不一样，[LINK]只能出现在文档的 HEAD 节中，但其出现次数是任意的。

由于历史原因，浏览器支持违反 HTML 规范的页面，这是为了让那些老旧的、不规整的页面也能够浏览，但在处理样式表方面，Internet Explorer 和 Firefox 都要求 Web 开发社区遵循规范。这就导致即使要承担降低用户体验的风险，违反了规范的页面（将 LINK 放到 HEAD 节的外面）仍然能够呈现。

在努力改善 Web 上最频繁访问的页面时，Yahoo!的门户团队最初将样式表放到了页面底部，结果适得其反。他们发现最佳解决方案就是遵循 HTML 规范，将它放在顶部。两种情况——白屏和无样式内容的闪烁——都不再是风险。如果你的样式表不要求呈现页面，可以想办法在文档加载完毕后动态加载进来，如第 8 章中“加载后下载”一节所述。否则，不管你的样式表在呈现页面时是否必需，都应该遵守这个规则。

使用 LINK 标签将样式表放在文档 HEAD 中

www.TopSage.com

第 6 章

规则 6——将脚本放在底部
Rule 6: Put Scripts at the Bottom

第 5 章介绍了将样式表放在页面的底部会阻碍页面逐步呈现，以及如何通过将其移至文档的 HEAD 中来解决这一问题。脚本（外部 JavaScript 文件）会引起类似的问题，但解决方案恰好相反——最好将脚本从页面的顶部移到底部（如果可以的话）。这样页面既可以逐步呈现，也可以提高下载的并行度。首先通过一个示例来看看这些问题。

脚本带来的问题
Problems with Scripts

演示该问题最好的方法是使用一个在页面中部放置脚本的示例。

将脚本放在中部的示例

http://stevesouders.com/hpws/js-middle.php

经过编程的脚本下载需要 10 秒钟，因此很容易看到问题——页面的下半部分要花大约 10 秒才能显示出来（第 5 章“sleep.cgi”一节解释了如何配置组件实现特定的加载时间）。出现这一现象是因为脚本阻塞了并行下载。在回顾了浏览器如何并行下载之后，我们再回过头解决这一问题。

示例页面的另一个问题是逐步呈现。在使用样式表时，页面逐步呈现会被阻止，直到所有的样式表下载完成。这就是最好将样式表移到文档的 HEAD 中的原因，这样就能首先下载它们而不会阻止页面呈现。使用脚本时，对于所有位于脚本以下的内容，逐步呈现都被阻塞了。将脚本放在页面越靠下的地方，意味着越多的内容能够逐步地呈现。

www.TopSage.com

大家网 TopSage.com

并行下载

Parallel Downloads

对响应时间影响最大的是页面中组件的数量。当缓存为空时，每个组件都会产生一个 HTTP 请求，有时即便缓存是完整的亦是如此。要知道浏览器会并行地执行 HTTP 请求，你可能会问，为什么 HTTP 请求的数量会影响响应时间呢？浏览器不能一次将它们都下载下来吗？

对此的解释要回到 HTTP 1.1 规范，该规范建议浏览器从每个主机名并行地下载两个组件（*http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1.4*）。很多 Web 页面需要从一个主机名下载所有的组件。查看这些 HTTP 请求会发现它们是呈阶梯状的，如图 6-1 所示。

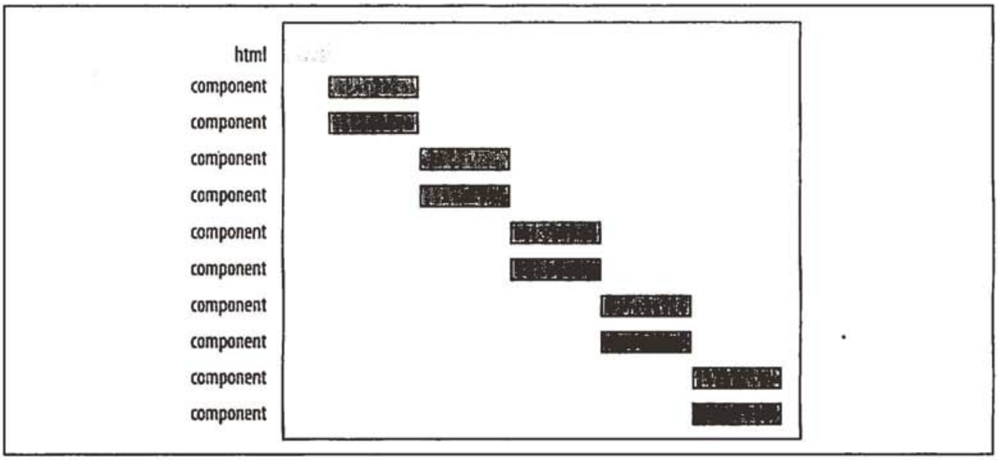


图 6-1：并行地下载两个组件

如果一个 Web 页面平均地将其组件分别放在两个主机名下，整体响应时间将可以减少大约一半。HTTP 请求的行为看起来会是图 6-2 所示的样子，可以并行下载 4 个组件（每个主机名两个）。为了对页面加载变快的现象给出可视的效果，其中每个时间块的横向宽度和图 6-1 是一样的。

每个主机名并行下载两个组件的限制只是一个建议。默认情况下，Internet Explorer 和 Firefox 都遵守这一建议，但用户可以重写该默认设置。Internet Explorer 将这个值存放在 Registry Editor 中（注 1）。

在 Firefox 中，可以使用 *about:config* 页面中的 `network.http.max-persistent-connections-per-server` 设置来修改这一默认设置。有意思的是，对于 HTTP 1.0，Firefox 的默认值是

注 1：关于如何重写改默认值，请参见 Microsoft 的 Web 文章"How to configure Internet Explorer to have more than two download sessions"，*http://support.micorosft.com/?kbid=282402*。

www.TopSage.com

大家网 TopSage.com

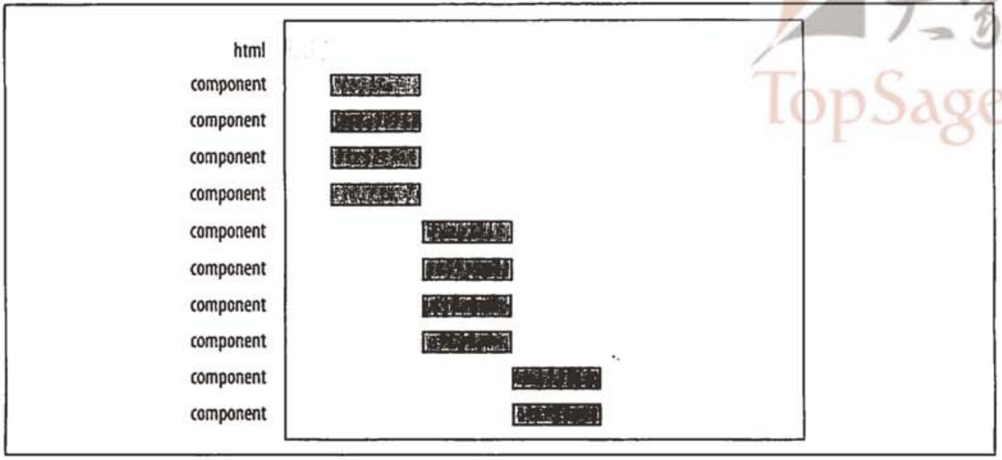


图 6-2：并行下载四个组件

每个主机名并行下载 8 个组件。图 6-3 展示了 Firefox 针对 HTTP 1.0 的设置在这个示例页面上产生的最短的响应时间。尽管只使用了一个主机名，但它比图 6-2 所示的结果更好。

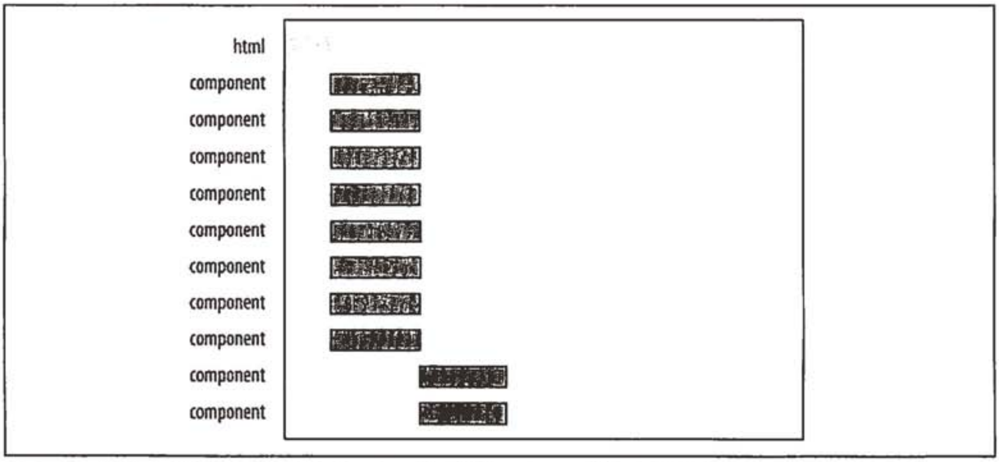


图 6-3：并行下载八个组件（Firefox 针对 HTTP 1.0 的默认设置）

今天的很多网站使用 HTTP 1.1，但将并行下载数增加到每个主机名超过两个也是有可能的。前端工程师与其依赖用户来修改浏览器设置，不如简单地使用 CNAME（DNS 别名）来将组件分别放到多个主机名中。但增加并行下载数量并不是没有开销的，其优劣取决于

www.TopSage.com

你的带宽和 CPU 速度，过多的并行下载反而会降低性能。Yahoo!的研究表明，使用两个主机名比使用 1、4 或 10 个主机名能带来更好的性能（注 2）。

大家网 TopSage.com

脚本阻塞下载
Scripts Block Downloads

并行下载组件的优点是很明显的。然而，在下载脚本时并行下载实际上是被禁用的——即使使用了不同的主机名，浏览器也不会启动其他的下载。其中一个原因是，脚本可能使用 `document.write` 来修改页面内容，因此浏览器会等待，以确保页面能够恰当地布局。

在下载脚本时浏览器阻塞并行下载的另外一个原因是为了保证脚本能够按照正确的顺序执行。如果并行下载多个脚本，就无法保证响应是按照特定顺序到达浏览器的。例如，后面的脚本比页面中之前出现的脚本更小，它可能首先执行。如果它们之间存在着依赖关系，不按顺序执行就会导致 JavaScript 错误。下面的例子演示了脚本是如何阻塞并行下载的。

脚本阻塞下载的示例

http://stevesouders.com/hpws/js-blocking.php

该页面按顺序包含下列组件——

1. 来自 host1 的一个图片
2. 来自 host2 的一个图片
3. 来自 host1 的一个加载需要 10 秒钟的脚本
4. 来自 host1 的一个图片
5. 来自 host2 的一个图片

按照之前对浏览器并行下载的介绍，你会期望在下载来自 host1 的前两个组件的同时，能够下载来自 host2 的两个图片。图 6-4 展示了实际的情况。

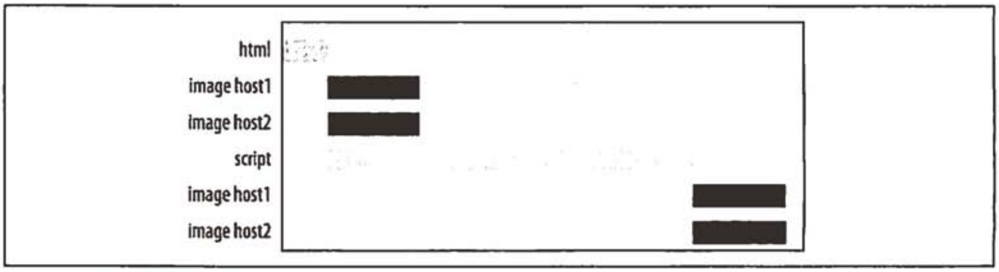


图 6-4：脚本阻塞下载

注 2：Tenni Theurer，"Performance Research, Part 4: Maximizing Parallel Downloads in the Carpool Lane"，*http://yuiblog.com/blog/2007/04/11/performance-research-part-4/*。

www.TopSage.com

Internet Explorer 和 Firefox 浏览器都是从下载来自 host1 和 host2 的第一个图片开始的。然后下载来自 host1 的脚本。此时发生了非预期的行为。在下载脚本时（这里为了说明问题将其夸张到需要 10 秒钟），来自 host1 和 host2 的第二个图片的下载被阻塞了。直到脚本加载完毕，才开始下载页面中的剩余组件。

TopSage.com

最差情况：将脚本放在顶部

Worst Case: Scripts at the Top

至此，脚本对 Web 页面的影响就清楚了：

- 脚本会阻塞对其后面内容的呈现。
- 脚本会阻塞对其后面组件的下载。

如果将脚本放在页面顶部——正如通常的情况那样——页面中的所有东西都位于脚本之后，整个页面的呈现和下载都会被阻塞，直到脚本加载完毕。请试一下下面的示例。

将脚本放在顶部的示例

http://stevesouders.com/hpws/js-top.php

由于整个页面的呈现被阻塞，因此导致了第 5 章所介绍的白屏现象。逐步呈现对于良好的用户体验来说是非常重要的，但缓慢的脚本下载延迟了用户所期待的反馈。此外，由于并行下载数的减少，不管图片显示有多快，都要被延迟。图 6-5 展示了这个页面中组件下载得比所期望的要慢。

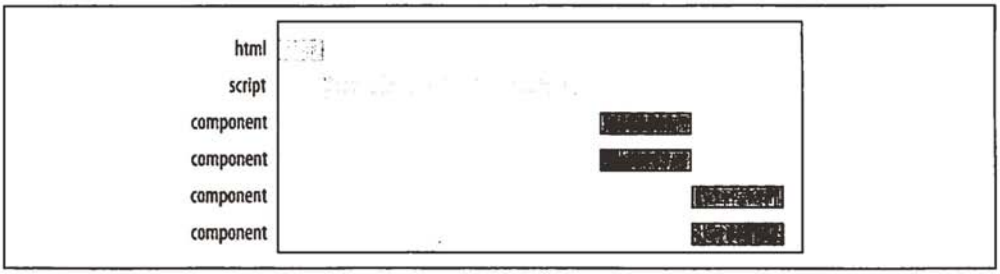


图 6-5：将脚本放在顶部阻塞了整个页面

最佳情况：将脚本放在底部

Best Case: Scripts at the Bottom

放置脚本的最好地方是页面的底部。这不会阻止页面内容的呈现，而且页面中的可视组件可以尽早下载。图 6-6 表明，在将脚本放在底部后，虽然其请求时间较长，但对页面的影响很小。通过访问下面的示例可以看到这一点。

www.TopSage.com

大家网 TopSage.com

将脚本放在底部的情况

http://stevesouders.com/hpws/js-bottom.php

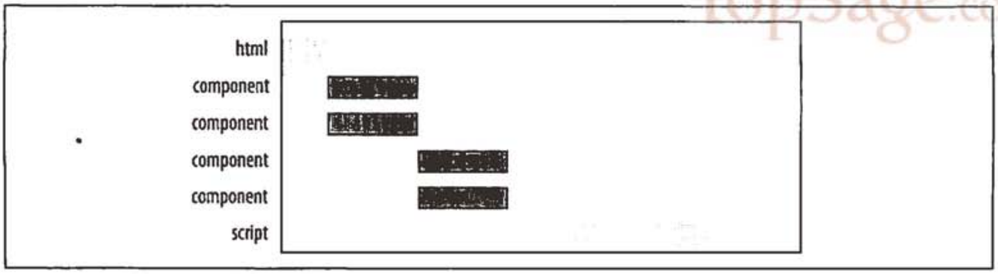


图 6-6：将脚本放在底部能够产生较小的影响

把两个页面——脚本放在顶部的和脚本放在底部的——并列放在一起浏览，其对比更为突出。可以在下面这个示例中看到这一点。

顶部脚本 VS 底部脚本的示例

http://stevesouders.com/hpws/move-scripts.php

正确地放置
Putting It in Perspective

前面那些示例是使用了需要 10 秒才能下载完的脚本。希望你使用的脚本不需要这么长时间的延迟，但一个脚本很可能花费比预期长的时间，用户的带宽也会影响脚本的响应时间。你的页面中的脚本所产生的影响可能没有这里展示的这么严重，但仍需要注意。在页面中包含多个脚本也会带来问题。

在很多情况下，很难将脚本移到底部。例如，如果脚本使用 `document.write` 向页面中插入了内容，就不能将其移动到页面中靠后的位置。此外还会有作用域问题。很多情况下，可用其他办法解决这些情形。

经常出现的另外一种建议是使用**延迟**（*Defferred*）脚本。`DEFER` 属性表明脚本不包含 `document.write`，浏览器得到这一线索就可继续进行呈现。从下面的示例可以看到这一点。

延迟脚本的示例

http://stevesouders.com/hpws/js-defer.php

不幸的是，在 Firefox 中，即便是延迟脚本也会阻塞呈现和并行下载。在 Internet Explorer 中，页面中靠后的组件下载得明显要晚一些。如果一个脚本可以延迟，那么它一定可以移到页面的底部。这是加速 Web 页面的最佳方式。

将脚本移到页面底部

www.TopSage.com

规则 7——避免 CSS 表达式

Rule 7: Avoid CSS Expressions

CSS 表达式是动态设置 CSS 属性的一种强大（并且危险）的方式。它受到 Internet Explorer 版本 5 和之后版本的支持。我们从一个传统的设置背景色的 CSS 规则开始——

```
background-color: #B8D4FF;
```

对于很多动态页面，可以使用 CSS 表达式将背景色设置为每小时变化一次。

```
background-color: expression( (new Date()).getHours()%2 ? "#B8D4FF" : "#F08A00" );
```

正如这里所示，expression 方法接受一个 JavaScript 表达式。CSS 属性将被设置为对 JavaScript 表达式进行求值的结果。

expression 方法被其他浏览器简单地忽略了，但是对于 2E 而言这是一种有用的工具，能够在 Internet Explorer 中设置属性，创建跨浏览器的一致体验。例如，Internet Explorer 不支持 min-width 属性。CSS 则是解决这一问题的一种方法。下面的示例确保一个页面的宽度最少是 600 像素，Internet Explorer 可以识别表达式，而其他浏览器识别静态设置：

```
width: expression( document.body.clientWidth < 600 ? "600px" : "auto" );
min-width: 600px;
```

大多数浏览器会忽略 width 属性而使用 min-width 属性，因为它们不支持 CSS 表达式。Internet Explorer 则忽略 min-width，而根据文档的宽度动态地设置 width 属性。当页面变化时，CSS 表达式会重新求值，如大小改变时。这确保了每当用户改变其浏览器大小时，宽度都能恰当地调整。对 CSS 表达式的频繁求值使其得以工作，但也导致 CSS 表达式的低下性能。

www.TopSage.com

大家网
TopSage.com

更新表达式

Updating Expressions

表达式的问题在于对其进行的求值的频率比人们期望的要高。它们不只在页面呈现和大小改变时求值，当页面滚动、甚至用户鼠标在页面上移过时都要求值。我们可以向 CSS 表达式中添加一个计数器来进行跟踪，看看 CSS 表达式何时被求值，以及有多么频繁。

表达式计数器的示例

http://stevesouders.com/hpws/expression-counter.php

CSS 表达式计数器示例使用了下面的 CSS 规则——

```
P {
  width: expression( setCntr(), document.body.clientWidth<600 ? "600px" : "auto" );
  min-width: 600px;
  border: 1px solid;
}
```

`setCntr()`函数增加一个全局变量的值并将这个值写到页面中的一个文本框里。页面中有 10 个段落。加载页面会执行 CSS 表达式 40 次。这之后，对于各种事件，如改变大小、滚动和鼠标移动，该 CSS 表达式都会被求值 10 次。在页面上来回移动鼠标可以很轻易地产生 10000 次以上的求值。在这个例子里 CSS 的危险显而易见。最郁闷的是，在 Internet Explorer 中单击这个页面中的文本框之后，你就不得不终止这个进程了。

围绕问题展开工作

Working Around the Problem

很多 CSS 专家都非常熟悉 CSS 表达式，也知道如何避免前面的例子中显现出来的缺陷。有两种技术可以避免 CSS 表达式产生这一问题——创建一次性表达式和使用事件处理器取代 CSS 表达式。

一次性表达式

One-Time Expressions

如果 CSS 表达式必须被求值一次，那么可以在这一次执行中重写它自身。本章开始时定义的背景样式就非常适合使用这种方式——

```
<style>
P {
   background-color: expression( altBgcolor(this) );
}
</stle>

<script type = "text/javascript"
```


www.TopSage.com

```
function altBgcolor(elem) {
    elem.style.backgroundColor = (new Date()).getHours()%2 ? "#F08A00" : "#B8D4FF";
}
</script>
```

CSS 表达式调用了 altBgcolor() 函数，而该函数将样式的 background-color 属性设置为一个明确的值，并移除了 CSS 表达式。这个样式被关联到页面中的 10 个段落。无论在改变大小、滚动或在页面上移动鼠标之后，CSS 表达式都只会执行 10 次，这比前一个例子中的上万次要好很多。

一次性表达式的示例

http://stevesouders.com/hpws/onetime-expression.php

事件处理器

Event Handlers

我看到过的绝大多数使用 CSS 表达式的情形，都可以找到不需要 CSS 表达式的替代方法。CSS 表达式从自动绑定到浏览器事件中获益，但这也是它的缺陷。除了使用 CSS 表达式之外，前端工程师还可以尝试使用事件处理器来为特定的事件提供所期望的动态行为。这就避免了在无关事件发生时对表达式的求值。下面的事件处理器示例展示了通过在 onresize 事件中设置样式的 width 属性来修正 min-width 问题，避免了鼠标移动、滚动等事件产生的成千上万次不必要的求值。

事件处理器的示例

http://stevesouders.com/hpws/event-handler.php

当浏览器的大小改变时，这个例子使用 setMinWidth() 函数来修改所有段落元素的大小——

```
function setMinWidth() {
    setCntr();
    var aElements = document.getElementsByTagName("p");
    for ( var i = 0; i < aElements.length; i++ ) {
        aElements[i].runtimeStyle.width = ( document.body.clientWidth<600 ?
"600px" : "auto" );
     }
}

if ( 1 != navigator.userAgent.indexOf("MSIE") ) {
    window.onresize = setMinWidth;
}
```

这会在浏览器大小改变时动态地设置宽度，但在第一次呈现时这并不能恰当地设置段落的大小。因此，页面还需要使用“一次性表达式”一节中介绍的方法，通过 CSS 表达式设置初始宽度，并在第一次求值后重写 CSS 表达式。

www.TopSage.com

大家网
TopSage.com

小结
Conclusion

有些规则用于处理页面加载之后的性能问题，本章所介绍的只是其中之一，这通常是由CSS表达式引起的问题。然而，有的时候，CSS表达式也会影响页面的加载时间。Yahoo!的一项功能使用了CSS表达式，导致页面首次呈现时间延迟了20秒。这个结果不是预期的，而是需要花一些时间进行诊断。类似的，谁会想到用户单击了一个文本框会导致Internet Explorer锁死呢？关于复杂的CSS不兼容性——如 `min-width` 和 `location: fixed`——的完整讨论超出了本书的范围，但可以明确的是——在没有深入了解底层影响的情况下使用CSS表达式是很危险的。

避免 CSS 表达式

www.TopSage.com

规则 8——使用外部 JavaScript 和 CSS

Rule 8: Make JavaScript and CSS External

本书介绍的很多性能规则用于处理如何管理外部组件，如通过 CDN 提供外部组件（规则 2）、确保其拥有长久的 `Expires` 头（规则 3）以及压缩其内容（规则 4）。然而，在出现这种考虑之前，我们应该问一个更基本的问题——JavaScript 和 CSS 是应该包含在外部文件中还是内联在页面中？正如我们将要看到的那样，通常使用外部文件更好。

内联 VS 外置

Inline vs. External

我们首先来对比一下内联 JavaScript 和 CSS 与将其外置之间的区别。

纯粹而言，内联快一些

In Raw Terms, Inline Is Faster

我编写了两个示例，用于演示内联 JavaScript 和 CSS 可以产生比外部文件更快的响应速度。

内联 JS 和 CSS 的示例

http://stevesouders.com/hpws/inlined.php

外部 JS 和 CSS 的示例

http://stevesouders.com/hpws/external.php

内联示例只有一个 HTML 文档，其大小为 87KB，所有的 JavaScript 和 CSS 都包含在 HTML 文件自身中。外部示例包含一个 HTML 文档(7KB)、一个样式表(59KB)和三个脚本(1KB、11KB 和 9KB)，总计 87KB。尽管所需下载的总数据量是相同的，内联示例还是比外部示例快 30%~50%。这主要是因为外部示例需要承担多个 HTTP 请求带来的开销（第 1 章介绍了减少 HTTP 请求数量的重要性）。尽管外部示例可以从样式表和脚本的并行下载中获益，但一个 HTTP 请求与五个 HTTP 请求之间的差距导致内联示例更快一些。

www.TopSage.com

尽管结果如此，现实中还是使用外部文件会产生较快的页面。这是由于本例中没有涉及的、外部文件所带来的收益——JavaScript 和 CSS 文件有机会被浏览器缓存起来。HTML 文档——至少是那些包含动态内容的 HTML 文档——通常不会被配置为可以进行缓存。当遇到这种情况时（HTML 文档没有被缓存），每次请求 HTML 文档都要下载内联的 JavaScript 和 CSS。另一方面，如果 JavaScript 和 CSS 是外部文件，浏览器就能缓存它们，HTML 文档的大小减小，而且不会增加 HTTP 请求的数量。

关键因素是，与 HTML 文档请求数量相关的、外部 JavaScript 和 CSS 组件被缓存的频率。这个因素尽管难以量化，但可以通过下面的手段进行衡量。

页面查看
Page Views

每个用户产生的页面查看越少，内联 JavaScript 和 CSS 的论据越强势。想象一个普通用户每个月只访问你的网站一次。在每次访问之间，外部 JavaScript 和 CSS 文件很有可能从浏览器的缓存中移除，即使该组件有一个长久的 `Expires` 头（有关使用长久 `Expires` 头的更多内容，请参见第 3 章）。

另一方面，如果普通用户能够产生很多的页面查看，浏览器很可能将（具有长久的 `Expires` 头的）外部文件放在其缓存中。使用外部文件提供 JavaScript 和 CSS 带来的收益会随着每用户每月的页面查看次数或每用户每会话产生的页面查看次数的增长而增加。

空缓存 VS 完整缓存
Empty Cache vs. Primed Cache

在比较内联和外部文件时，知道用户缓存外部组件的可能性这一点非常重要。我们在 Yahoo!进行了测量，发现每天至少携带完整缓存访问 Yahoo!功能一次的用户占 40%~60%（注 1）。同样的研究表明，具有完整缓存的页面查看数量占 75%~85%。注意第一个统计测量的是“唯一用户”而第二个是“页面查看”。具有完整缓存的页面查看所占的百分比比携带完整缓存的唯一用户的百分比高，这是因为很多用户在一次会话中进行了多次页面查看。每天，用户可能只有开始的一次访问携带的是空缓存，之后的多次后续页面查看都具有完整缓存。有关这一研究的更多信息，请参见第 3 章。

注 1：Tenni Theurer，“Performance Research, Part 2: Browser Cache Usage - Exposed!”，*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*。

www.TopSage.com

这些度量结果随着网站的类型而变。知道这些统计值有助于评估使用外部文件相对于使用内联带来的潜在收益。如果你的网站的本质上能够为用户带来高完整缓存率，使用外部文件的收益就更大。如果不大可能产生完整缓存，则内联是更好的选择。

TopSage.com

组件重用

Component Reuse

如果你的网站中的每个页面都使用了相同的 JavaScript 和 CSS，使用外部文件可以提高这些组件的重用率。在这种情况下使用外部文件更加具有优势，因为当用户在页面间导航时，JavaScript 和 CSS 组件已经位于浏览器的缓存中了。

相反的情况也很容易理解——如果没有任何两个页面共享相同的 JavaSript 和 CSS，重用率就会非常低。难的是绝大多数网站不是非黑即白的。这就带来一个单独但相关的问题——当把 JavaScript 和 CSS 打包到外部文件中时，应该把边界划在哪里？

争论在文件越少越好（详细分析请参见第 1 章）这个前提下展开。在典型情况下，页面之间 JavaScript 和 CSS 的重用既不可能 100%重叠，也不可能 100%无关。在这种中间情形中，一个极端就是为每个页面提供一组分离的外部文件。这种方式的缺点在于，每个页面都强制用户使用另外一组外部组件并产生令响应时间变慢的 HTTP 请求。这种方式对于普通用户只访问一个页面和很少进行跨页访问的网站来说是有意义的。

另一个极端是创建一个单独的、联合了所有 JavaScript 的文件，再创建一个包含所有 CSS 的文件。这只要求用户生成一个 HTTP 请求，但它增加了用户首次进行页面查看时的下载数据量。在这种情况下，用户浏览页面时要下载的 JavaScript 和 CSS 要多于所需的数量。而且，在任何一块独立的脚本或样式表改变后，都需要更新这个文件，使所有用户已经缓存了的当前版本无效。这种情况对于那些每用户每月会话数量较高、普通用户在一个会话中访问多个不同页面的网站来说是有意义的。

如果你的网站并不符合这两种极端情况，最好的答案就是折中。将你的页面划分成几种页面类型，然后为每种类型创建单独的脚本和样式表。这比维护一个单独的文件要复杂，但通常比为每个页面维护不同的脚本和样式表要容易，并且对于给定的任意页面都只需下载很少的多余的 JavaScript 和 CSS。

最后，你作出的与 JavaScript 和 CSS 外部文件的边界相关的决定影响着组件的重用程度。

www.TopSage.com

如果你可以找到一个平衡点，实现较高的重用度，则将 JavaScript 和 CSS 部署到外部文件的论据更加强势一些。如果重用度很低，还是内联更有意义一些。

TopSage.com

典型的对比结果
Typical Results in the Field

在对内联和使用外部文件进行对比分析时，关键点在于与 HTML 文档请求数量相关的、外部 JavaScript 和 CSS 组件被缓存的频率。在前一节里，我介绍了三种基准（页面查看、空缓存 VS 完整缓存和组件重用），这有助于你确定最好的选择。对于任何网站来说，正确答案都依赖于这些基准。

很多网站都处于这些基准的中间位置。他们拥有每用户每月 5~15 次页面查看，并有每用户每会话 2~5 次页面查看。空缓存浏览的情况和 Yahoo!类似——每天有 40%~60%的唯一用户具有完整缓存，每天有 75%~85%的页面查看是在完整缓存下进行的。页面之间有着恰当数量的 JavaScript 和 CSS 重用，使得少数几个文件即可覆盖每一种主要的页面类型。

对于具备这样的度量结果的网站，最好的解决方案通常是将 JavaScript 和 CSS 部署到外部文件中。下面的示例演示了这一点，其中的外部组件可以被浏览器缓存。反复加载这一页面，并与第一个示例“内联 JS 和 CSS”的结果进行比较，可以看出使用带有长久 Expires 头的外部文件是最快的方式。

可缓存的外部 JS 和 CSS 的示例

http://stevesouders.com/hpws/external-cacheable.php

主页
Home Pages

我所见过的使用内联方式反而更好的一个例外是主页。主页是选择作为浏览器默认页的 URL，如 Yahoo!的主页（*http://www.yahoo.com*）和 My Yahoo!（*http://my.yahoo.com*）。我们来从主页的角度看一下这三个基准——

页面查看

主页拥有每月很高的页面查看数量。确切地说，只要浏览器打开，主页就被访问了。然而，通常每个会话只有一个页面查看。

空缓存 VS 完整缓存

完整缓存的百分比要比其他网站更低。出于安全的原因，很多用户选择在每次关闭浏览器时清空缓存。下一次用户打开浏览器时，产生的是一个到主页的空缓存页面查看。

www.TopSage.com

大家网 TopSage.com

组件重用

> 重用率很低。很多主页是用户来到网站后访问的唯一一个页面，因此它们谈不上重用。

分析了这些基准之后，我们更加倾向使用内联而不是外部文件。主页倾向于使用内联还有很多因素——它们对响应能力有着更高的要求，即便是在空缓存场景中。如果一个公司决定启动一项战略，鼓励用户将该公司网站设置为他们的主页，这些公司最不愿意看到的就是一个缓慢的主页。为了公司的主页战略能够成功，页面必须很快。

没有适用于所有主页的唯一答案。但在讨论到主页时必须评估这里给出的因素。如果正确答案是内联，则你可以在下一节中找到有用的信息。下一节介绍了两种技术，能够利用外部文件得到内联的收益（如果可能的话）。

两全其美
The Best of Both Worlds

即便所有的因素都偏向于内联，将所有 JavaScript 和 CSS 都添加到页面中还是会感觉很低效，而且无法利用浏览器的缓存。这里介绍的两项技术使你既可以获得内联的优势，同时也能缓存外部文件。

加载后下载
Post-Onload Download

有一些主页——如 Yahoo!主页和 My Yahoo!——通常每会话只有一个页面查看。然而，并不是所有主页都这样。Yahoo! Mail 就是一个很好的例子，其主页通常伴随着后续页面查看（在初始页之后，还要访问其他页面，如查看或编写 Email 消息）。

对于作为多次页面查看中的第一次的主页，我们希望为主页内联 JavaScript 和 CSS，但又能为所有后续页面查看提供外部文件。这可以通过在主页加载完成后动态下载外部组件来实现（通过 onload 事件）。这能够将外部文件放到浏览器的缓存中以便用户接下来访问其他页面。

加载后下载的示例

> *http://stevesouders.com/hpws/post-onload.php*

“加载后下载”示例的 JavaScript 代码将 `doOnload` 函数关联到文档的 `onload` 事件。在 1 秒的延迟之后（确保页面呈现完毕），就会下载所需的 JavaScript 和 CSS 文件。这通过创建对应的 DOM 元素（分别是 `script` 和 `link`）并赋予指定的 URL 来实现——

www.TopSage.com

大家网 TopSage.com

```
<script>
function doOnload() {
    setTimeout("downloadComponents()", 1000);
}

window.onload = doOnload;

// Download external components dynamically using JavaScript.
function downloadComponents() {
    downloadJS("http://stevesouders.com/examples/testsma.js");
    downloadCSS("http://stevesouders.com/examples/testsm.css");
}

// Download a script dynamically.
function downloadJS(url) {
    var elem = document.createElement("script");
    elem.src = url;
    document.body.appendChild(elem);
}

// Download a stylesheet dynamically.
function downloadCSS(url) {
    var elem = document.createElement("link");
    elem.rel = "stylesheet";
    elem.type = "text/css";
    elem.href = url;
    document.body.appendChild(elem);
}
</script>
```

在这些页面中，JavaScript 和 CSS 被加载到页面中两次（先是内联的，然后是外部的）。要使其能够工作，必须处理双重定义。例如脚本，可以定义但不能执行任何函数（至少不能让用户察觉）。使用了相对单位（百分比或 em）的 CSS 如果指定两次可能会产生问题。将这些组件放到一个不可见的 IFrame 中是一种更好的方式，能够避免这些问题。

动态内联
Dynamic Inlining

如果主页服务器知道一个组件是否在浏览器的缓存中，它可以在内联或使用外部文件之间做出最佳的选择。尽管服务器不能查看浏览器缓存中有些什么，但可以用 cookies 做指示器。如果 cookie 不存在，就内联 JavaScript 和 CSS。如果 cookie 出现了，则有可能外部组件位于浏览器的缓存中，并使用了外部文件。“动态内联”示例展示了这一技术。

www.TopSage.com

动态内联的示例

http://stevesouders.com/hpws/dynamic-inlining.php

由于每个用户开始的时候都没有 cookie，因此必须有一种途径来引导这一过程。这可以通过使用前一个例子中的加载后下载技术来完成。当用户第一次访问页面时，服务器发现没有 cookie，于是生成一个内联了组件的页面。然后服务器添加 JavaScript 来在页面加载后动态下载外部文件（并设置 cookie）。下一次访问页面时，服务器看到了 cookie，就会生成一个使用外部文件的页面。

下面给出了处理这种动态行为的 PHP 代码——

```
<?php
if ( $_COOKIE["CA"] ) {
   // If the cookie is present, it's likely the component is cached.
   // Use external files since they'll just be read from disk.
   echo <<<OUTPUT
<link rel="stylesheet" href="testsm.css" type="text/css">
<script src="testsma.js"></script>
OUTPUT;
}
else {
   // If the cookie is NOT present, it's likely the component is NOT cached.
   // Inline all the components and trigger a post-onload download of the files.
   echo "<style>\n" . file_get_contents("testsm.css") . "</style>\n";
   echo "<script type=\"text/javascript\">\n" . file_get_contents("testsma.js") .
"</script>\n";
   // Output the Post-Onload Download JavaScript code here.
   echo <<<ONLOAD
<script>
function doOnload() {
   setTimeout("downloadComponents()", 1000);
}

window.onload = doOnload;

// Download external components dynamically using JavaScript.
function downloadComponents() {
   document.cookie = "CA=1";
[snip...]
ONLOAD;
}
?>
```

我没有给出所有“加载后下载”的 JavaScript 代码（使用“[snip...]”代替），因为这已经在前面的“加载后下载”一节中包含了。然而，我展示了足够的代码来解释 `downloadComponents` 函数是如何设置 `CA` cookie 的。这是唯一的改动，但也是在后续页面查看中利用缓存的关键。

www.TopSage.com

这种方式的美好之处在于它的宽容。即便 cookie 的状态和缓存的状态不匹配，页面也能够工作，只是没有本应该的那么优化而已。基于会话的 cookie 技术在内联时会发生错误，即便组件已经被放到浏览器缓存中了——如果用户重新打开浏览器，基于会话的 cookie 会消失，但组件依然存在于缓存中。将 cookie 从基于会话的改为短期的（数小时或数天）可以解决这个问题，但当它们并不在浏览器缓存中时，使用外部文件又会出错。不管出现哪种情况，页面都能够工作，并且从全体用户的角度来看，它能够在内联和外部文件之间做出智能的选择，从而改善响应时间。

将 JavaScript 和 CSS 放到外部文件中

www.TopSage.com

规则 9——减少 DNS 查找
Rule 9: Reduce DNS lookups

Internet 是通过 IP 地址来查找服务器的。由于 IP 地址很难记忆，通常使用包含主机名的 URL 来取代它，但当浏览器发送其请求时，IP 地址仍然是必需的。这就是 Domain Name System（DNS）所处的角色。DNS 将主机名映射到 IP 地址上，就像电话本将人名映射到他们的电话号码一样。当你在浏览器中键入 *www.yahoo.com* 时，连接到浏览器的 DNS 解析器会返回服务器的 IP 地址。

这个解释强调了 DNS——URL 和实际宿主它们的服务器之间的一个间接层——的另外一项优点。如果一个服务器被另外一个具有不同 IP 地址的服务器替代了，DNS 允许用户使用同样的主机名来连接到新的服务器。或者，比如在 *www.yahoo.com* 的例子中，可以将多个 IP 地址关联到一个主机名，为网站提供高冗余度。

然而，DNS 也是开销。通常浏览器查找一个给定的主机名的 IP 地址要花费 20~120 毫秒。在 DNS 查找完成之前，浏览器不能从主机名那里下载到任何东西。响应时间依赖于 DNS 解析器（通常由你的 ISP 提供）、它所承担的请求压力、你与它之间的距离和你的带宽速度。在从浏览器的角度回顾完 DNS 的工作之后，我将介绍如何减少页面花在 DNS 查找上的时间。

DNS 缓存和 TTL
DNS Caching and TTLs

DNS 查找可以被缓存起来以提高性能。这种缓存可以发生在由你的 ISP 或局域网中的一台特殊的缓存服务器上，但我们这里要探索的是发生在独立用户的计算机上的 DNS 缓存。如图 9-1 所示，在用户请求了一个主机名之后，DNS 信息会留在操作系统的 DNS 缓存中（Microsoft Windows 上的“DNS Client 服务”），之后对于该主机名的请求将无需进行过多的 DNS 查找，至少短时间内不需要。

www.TopSage.com

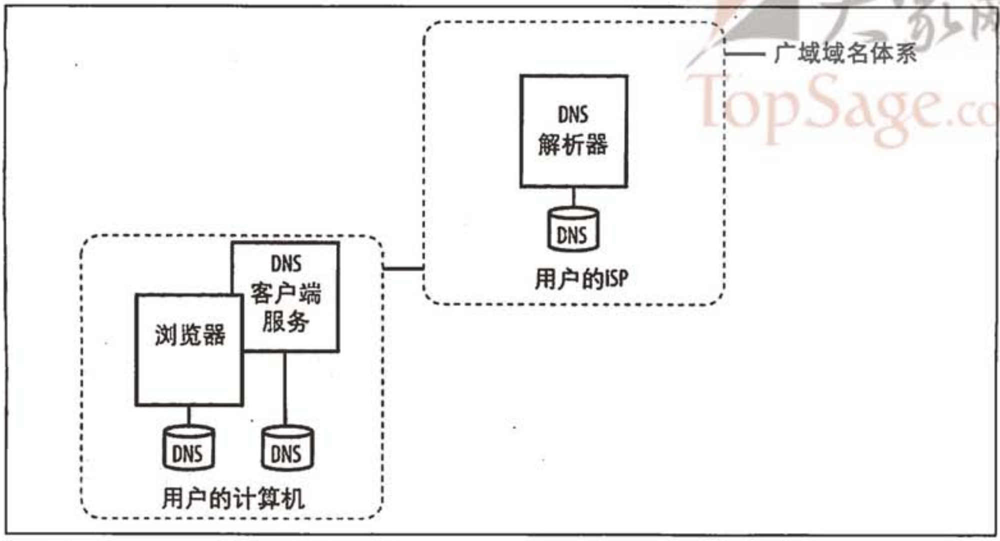


图 9-1：从浏览器角度看 DNS 缓存

够简单吧？等一下——很多浏览器拥有其自己的缓存，和操作系统的缓存相分离。只要浏览器在其缓存中保留了 DNS 记录，它就不会麻烦操作系统来请求这个记录。只有当浏览器缓存丢弃了记录时，它才会向操作系统询问地址——然后操作系统或者通过其缓存来响应这个请求，或者将请求发送给一台远程服务器，这时就会发生潜在的速度降低。

使事情更复杂的是，设计者知道 IP 地址会变化以及缓存会消耗内存。因此，应该周期性地清除缓存中的 DNS 记录，并通过大量不同的配置设置检测清除的频率有多高。

影响 DNS 缓存的因素

Factors Affecting DNS Caching

首先，服务器可以表明记录可以被缓存多久。查找返回的 DNS 记录包含了一个存活时间（Time-to-live，TTL）值。该值告诉客户端可以对该记录缓存多久。

尽管操作系统缓存会考虑 TTL 值，但浏览器通常忽略该值，并设置它自己的时间限制。此外，绪言 B 中讨论的 HTTP 协议中的 Keep-Alive 特性可以同时覆盖 TTL 和浏览器的时间限制。换句话说，只要浏览器和 Web 服务器愉快地通信着，并保持着 TCP 连接打开的状态，就没有理由进行 DNS 查找。

浏览器对缓存的 DNS 记录的数量也有限制，而不管缓存记录的时间。如果用户在短时间

www.TopSage.com

内访问了很多具有不同域名的网站，较早的 DNS 记录将被丢弃，必须重新查找该域名。

不过，要记得即便浏览器丢弃了 DNS 记录，操作系统可能依然保留着该记录，这能扭转一下局面，因为无需通过网络发送查询，从而避免了明显的延迟。

大家网 TopSage.com

TTL 值

TTL Values

十大美国网站发送给客户端的最大 TTL 值在 1 分钟到 1 小时之间，如表 9-1 所示。

表 9-1：TTL 值

域名	TTL
http://www.amazon.com	1 分钟
http://www.aol.com	1 分钟
http://www.cnn.com	10 分钟
http://www.ebay.com	1 小时
http://www.google.com	5 分钟
http://www.msn.com	5 分钟
http://www.myspace.com	1 小时
http://www.wikipedia.org	1 小时
http://www.yahoo.com	1 分钟
http://www.youtube.com	5 分钟

为什么这些值差距如此之大呢？这可能是由综合考虑和历史因素造成的。一个有趣的 RFC（注 1）详细介绍了 DNS 记录的格式和配置它们时常见的错误。其中第一条建议就是避免使用过短的 TTL 值，他的建议值是 1 天！

这些拥有巨大数量用户的顶级网站都在努力做到当服务器、虚拟 IP 地址（VIP）或联合定位掉线时提供快速故障转移。这也是 Yahoo!提供较短的 TTL 的原因。另一方面，MySpace 则定位到一个联合定位工具。对于其当前的网络拓扑，故障转移并不是很重要，因此他们选择了较长的 TTL，因为这可以减少 DNS 查找，同时也降低了其名称服务器的负载。

为 DNS 配置提供建议超出了本书的范围。而与本书有关的是，DNS 缓存是如何影响 Web 页面性能的。我们来从浏览器的角度观察一下 DNS 缓存，检查一下你的 Web 页面会产生多少 DNS 查找。

注 1："Common DNS Data File Configuration Errors"，*http://tools.ietf.org/html/rfc1537*。

www.TopSage.com

客户端收到的 DNS 记录的平均 TTL 值只有最大 TTL 值的一半。这是因为 DNS 解析器自身也拥有与 DNS 记录相关的 TTL。当浏览器进行 DNS 查找时，DNS 解析器返回的时间是其记录的 TTL 的剩余时间。如果最大 TTL 是 5 分钟，DNS 解析器返回的 TTL 范围可能是 1~300 秒，平均是 150 秒。对于给定的主机名，每次执行 DNS 查找时接收到的 TTL 值都会变化。

浏览器的视角

The Browser's Perspective

正如之前在“影响 DNS 缓存的因素”一节中讨论的那样，大量独立的变量决定了一个特定的浏览器在请求一个主机名时是否会进行远程 DNS 请求。尽管有 DNS 规范（*http://tools.ietf.org/html/rfc1034*），但在 DNS 缓存如何工作上，它给客户端留了较大的灵活性。我将着重介绍 Microsoft Windows 上的 Internet Explorer 和 Firefox，因为它们是最流行的平台。

Microsoft Windows 上的 DNS 缓存由 DNS Client 服务进行管理。你可以使用 `ipconfig` 命令来查看和刷新 DNS Client 服务——

```
ipconfig /displaydns
ipconfig /flushdns
```

重新启动也可以清空 DNS Client 服务缓存。除了 DNS Client 服务之外，Internet Explorer 和 Firefox 浏览器都有其自身的 DNS 缓存。重新启动浏览器会清空浏览器缓存，但不会清空 DNS Client 服务缓存。

Internet Explorer

Internet Explorer 的 DNS 缓存由三个注册表设置控制——`DnsCacheTimeout`、`KeepAliveTimeout` 和 `ServerInfoTimeOut`，可以创建在下面的注册表键下——

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\InternetSettings\
```

有两篇 Microsoft Support 文章（注 2）介绍了这些设置是如何影响 DNS 缓存的。这两篇文章提到这些设置的默认值如下——

- `DnsCacheTimeout`——30 分钟
- `KeepAliveTimeout`——1 分钟
- `ServerInfoTimeOut`——2 分钟

注 2：“How Internet Explorer uses the cache for DNS host entries”，*http://support.microsoft.com/default.aspx?scid=KB;en-us;263558*。
“How to change the default keep-alive time-out value in Internet Explorer”，*http://support.microsoft.com/kb/813827*。

www.TopSage.com

这暗示着（但没有很好地解释），如果 DNS 服务器 TTL 值小于 30 分钟的话，对浏览器进行 DNS 查找的频率产生的影响很小。一旦浏览器缓存了 DNS 记录，就会使用 30 分钟作为 TTL 值。如果发生了错误，刷新 DNS 查找就会比这要快，在正常情况下，很短的 TTL 值（30 分钟以下）在 Internet Explorer 中不会增加 DNS 查找的数量。

Keep-Alive 所扮演的角色也很重要。默认情况下，一个持久的 TCP 连接将会一直使用，直到其空闲 1 分钟为止。由于连接是持久的，因此无需 DNS 查找（绪言 B 中讨论了 Keep-Alive 的优点）。还有一个额外的优点——Keep-Alive 通过重用现有连接避免了重复的 DNS 查找。

`ServerInfoTimeOut` 的值为 2 分钟，说明尽管没有 Keep-Alive，如果一个主机名每两分钟重用了一次，并且没有发生错误，也无需进行 DNS 查找。在使用 Internet Explorer 的测试中，如果至少每两分钟重用一个主机名，可以超过 30 分钟不需要进行 DNS 查找（假设在访问其 IP 地址时没有发生过错误）。

当网络操作中心尝试通过 DNS 变化来转移流量时，这对他们来说是个重要的信息。如果一个 IP 上的流量已经被转移走，但该 IP 仍在运行，则使用旧的 DNS 记录的 Internet Explorer 用户至少需要 30 分钟才能更新 DNS。访问站点的活跃用户（至少每两分钟访问一次）会一直使用旧的 IP 地址而不会更新 DNS，直到发生错误。

Firefox

Firefox 介绍起来要更为简单。它具有下列配置设置——

- `network.dnsCacheExpiration`——1 分钟
- `network.dnsCacheEntries`——20
- `network.http.keep-alive.timeout`——5 分钟

DNS 记录的缓存时间比其 TTL 多 1 分钟。因为这个值很小，将 TTL 设置得较低（低于 1 小时）很可能增加在 Firefox 中访问页面所必需的 DNS 查找次数。

令人惊讶的是，默认情况下只有 20 条记录能够缓存在 Firefox 中。这意味着访问很多不同域名网站的用户将会因为 DNS 查找而比具有同样行为的、使用 Internet Explorer 的用户更慢。

Firefox 的 Keep-Alive 超时时间比 Internet Explorer 更长——5 分钟比 1 分钟。确保你的服务器支持 Keep-Alive，以减少用户导航到你的网站时所需的 DNS 查找。

www.TopSage.com

Fasterfox（*http://fasterfox.mozdev.org*）是一款著名的 Firefox 插件，用于测量和改善 Firefox 的性能。经过对比，Fasterfox 将 DNS 设置修改为下面的值——

大家网
TopSage.com

- `network.dnsCacheExpiration`——1 小时
- `network.dnsCacheEntries`——512
- `network.http.keep-alive.timeout`——30 秒

减少 DNS 查找
Reducing DNS Lookups

当客户端的 DNS 缓存为空（浏览器和操作系统都是）时，DNS 查找的数量与 Web 页面中唯一主机名的数量相等。这包括页面 URL、图片、脚本文件、样式表、Flash 对象等的主机名。减少唯一主机名的数量就可以减少 DNS 查找的数量。这方面的优秀示例是 Google（*http://www.google.com*），它的页面只需要一次 DNS 查找。

减少唯一主机名的数量会潜在地减少页面中并行下载的数量。避免 DNS 查找降低了响应时间，但减少并行下载可能会增加响应时间。正如第 6 章中“并行下载”一节中介绍的那样，一定数量的并行下载是好的，尽管增加了主机名的数量。对于 Google.com 来说，它的页面中只有两个组件。由于每个主机名可以并行下载两个组件，使用一个主机名既减少了 DNS 查找的数量，又最大化了并行下载。

今天的很多页面都有大量的组件——不会像 Google 那样。我的建议是将这些组件分别放到至少 2 个，但不要超过 4 个主机名下。这是在减少 DNS 查找和允许高度并行下载之间作出的很好的权衡。

绪言 B 中介绍的使用 Keep-Alive 所带来的好处在于，它可以通过重用现有连接，从而通过避免 TCP/IP 开销来减少响应时间。正如这里所介绍的，确保服务器支持 Keep-Alive 还能减少 DNS 查找，尤其是对 Firefox 用户来说。

通过使用 Keep-Alive 和较少的域名来减少 DNS 查找

www.TopSage.com

规则 10——精简 JavaScript
Rule 10: Minify JavaScript

JavaScript 作为一门解释型语言，是构建 Web 页面的首选。当以快速原型为基准开发用户界面时，解释语言要优于其他语言。由于没有了编译步骤，在最终部署前优化 JavaScript 的责任就落在了前端工程师身上。这个问题的一个方面——压缩——已经在第 4 章中讨论过了，这一章我将介绍在 JavaScript 部署过程中应该集成的另一个步骤——精简。

精简
Minification

精简（*Minification*）是从代码中移除不必要的字符以减小其大小，进而改善加载时间的实践。在代码被精简后，所有的注释以及不必要的空白字符（空格、换行和制表符）都将被移除。对于 JavaScript 而言，这可以改善响应时间效率，因为需要下载的文件大小减小了。

表 10-1 展示了十大美国网站中有多少使用了 JavaScript 精简——它们中的十分之四精简了其 JavaScript 代码。

表 10-1：十大网站的精简实践

网站	精简了外部脚本?
http://www.amazon.com	否
http://www.aol.com	否
http://www.cnn.com	否
http://www.ebay.com	是
http://froogle.google.com	是
http://www.msn.com	是
http://www.myspace.com	否
http://www.wikipedia.org	否
http://www.yahoo.com	是
http://www.youtube.com	否

www.TopSage.com

我们接下来看一下其他网站进行了精简后的。但首先，我需要谈一下另外一种更具挑战性的精简方式——混淆。

TopSage.com

混淆
Obfuscation

混淆（*Obfuscation*）是可以应用在源代码上的另外一种优化方式。和精简一样，它也会移除注释和空白，同时它还会改写代码。作为改写的一部分，函数和变量的名字将被转换为更短的字符串，这时的代码更加精炼，也更难阅读。通常这样做是为了增加对代码进行反向工程的难度，但这对提高性能也有帮助，因为这比精简更能减小代码的大小。

如果你的目的不是抵制反向工程，就会遇到是使用精简还是使用混淆的问题。精简是一个安全并且相当简单的过程。而混淆则更为复杂一些。混淆 JavaScript 有三个主要的缺点——

缺陷

　　由于混淆更加复杂，混淆过程本身很有可能引入错误。

维护

　　由于混淆会改变 JavaScript 符号，因此需要对任何不能改变的符号（例如 API 函数）进行标记，防止混淆器修改它们。

调试

　　经过混淆的代码很难阅读。这使得在产品环境中调试问题更加困难。

我从未见过精简会带来问题，但我见过由混淆引起的缺陷。要是像 Yahoo!这样维护庞大数量的 JavaScript，我的建议是使用精简而不是混淆。最终的决定需要考虑混淆能够带来的额外的代码大小减少量。在下一节中，我们将要进行一些实际的精简和混淆。

节省
The Savings

精简 JavaScript 代码的最流行的工具是 JSMin（*http://crockford.com/javascript/jsmin*），由我在 Yahoo!的一个同事 Douglas Crockford 开发。JSMin 的源代码以 C、C#、Java、JavaScript、Perl、PHP、Python 和 Ruby 等语言公开。JavaScript 领域中的工具选择不是很清楚。Dojo Compressor（已改名为 ShrinkSafe 并转移到 *http://dojotoolkit.org/docs/shrinksafe*）是我见过的用得最多的。为了完成比较，我将同时使用这两个工具。

www.TopSage.com

作为演示，我们将在来自 Yahoo! User Interface（YUI）库（*http://developer.yahoo.com/yui*）的 *event.js* 上使用这两个工具。该文件中的第一个函数的源代码如下——

```
YAHOO.util.CustomEvent = function(type, oScope, silent, signature) {
    this.type = type;
    this.scope = oScope || window;
    this.silent = silent;
    this.signature = signature || YAHOO.util.CustomEvent.LIST;
    this.subscribers = [];

    if (!this.silent) {
    }

    var onsubscribeType = "_YUICEOnSubscribe";
    if (type !== onsubscribeType) {
        this.subscribeEvent =
            new YAHOO.util.CustomEvent(onsubscribeType, this, true);
    }
};
```

同样的函数经过 JSMin 的处理之后，所有不必要的空白将被移除——

```
YAHOO.util.CustomEvent=function(type,oScope,silent,signature){this.type=type;this.
scope=oScope||window;this.silent=silent;this.signature=signature||YAHOO.util.
CustomEvent.LIST;this.subscribers=[];if(!this.silent){}
var onsubscribeType="_YUICEOnSubscribe";if(type!==onsubscribeType){this.subscribeEv
ent=new YAHOO.util.CustomEvent(onsubscribeType,this,true);}};
```

Dojo Compressor 移除了大部分的空白，同时还缩短了变量名。注意 `CustomEvent` 函数的第一个参数 `type` 被`_1` 替代了。

```
YAHOO.util.CustomEvent=function(_1,_2,_3,_4){
this.type=_1;
this.scope=_2||window;
this.silent=_3;
this.signature=_4||YAHOO.util.CustomEvent.LIST;
this.subscribers=[];
if(!this.silent){
}
var _5="_YUICEOnSubscribe";
if(_1!==_5){
this.subscribeEvent=new YAHOO.util.CustomEvent(_5,this,true);
}
};
```

表 10-2 展示了 6 家没有精简其 JavaScript 文件的公司可能获得的潜在节省。我下载了每个网站首页使用的 JavaScript 文件。表中列出了每个网站的 JavaScript 文件的原始大小，以及运行了 JSMin 和 Dojo Compressor 之后文件大小的减少量。平均下来，JSMin 可以使 JavaScript 文件的大小减小 21%，而 Dojo Compressor 可以达到 25%。

www.TopSage.com

表 10-2：使用 JSMin 和 Dojo Compressor 得到的大小减小量

网站	原始大小	JSMin 节省	Dojo Compressor 节省
http://www.amazon.com/	204KB	31KB（15%）	48KB（24%）
http://www.aol.com/	44KB	4KB（10%）	4KB（10%）
http://www.cnn.com/	98KB	19KB（20%）	24KB（25%）
http://www.myspace.com/	88KB	23KB（27%）	24KB（28%）
http://www.wikipedia.org/	42KB	14KB（34%）	16KB（38%）
http://www.youtube.com/	34KB	8KB（22%）	10KB（29%）
平均	85KB	17KB（21%）	21KB（25%）

什么时候混淆带来的额外节省能够抵得过它带来的额外风险呢？请看下面的 6 个示例，我将论证它们只需简单地精简其 JavaScript 代码，以此避免混淆可能引发的问题。其中一个例外是 Amazon，它使用混淆可以带来额外的 17KB（9%）的节省。在 Yahoo!只有具有高有效载荷（>100KB）的一小部分功能混淆了其 JavaScript 代码。正如我们在下面将要看到的那样，在结合了 gzip 压缩后，精简和混淆之间的差别将会减小。

示例
Examples

为了演示精简和混淆的优势，我生成了两个具有不同大小的脚本——一个小脚本（50KB）和一个大脚本（377KB）。小脚本在精简后可以减少到 13KB，混淆后可以减小到 12KB。大脚本精简后可以减小到 129KB，混淆后可以减小到 123KB。下面的六个示例测试了这两个脚本在三种状态下的结果。

一般的小脚本的示例

http://stevesouders.com/hpws/js-small-normal.php

经过精简的小脚本的示例

http://stevesouders.com/hpws/js-small-minify.php

经过混淆的小脚本的示例

http://stevesouders.com/hpws/js-small-obfuscate.php

一般的大脚本的示例

http://stevesouders.com/hpws/js-large-normal.php

经过精简的大脚本的示例

http://stevesouders.com/hpws/js-large-minify.php

经过混淆的大脚本的示例

http://stevesouders.com/hpws/js-large-obfuscate.php

www.TopSage.com

如表 10-3 所示，精简和混淆的效率几乎一样，但明显比一般情况快。对于小脚本，精简和混淆可以比一般情况快 100ms~110ms（17%~19%）。对于大脚本，精简和混淆可以比一般情况快 331ms~341ms（30%~31%）。

表 10-3：精简过和混淆过的脚本的响应时间

脚本大小	一般	经过精简	经过混淆
小（50K）	581ms	481ms	471ms
大（377K）	1092ms	761ms	751ms

正如前一节提到的，在结合使用了 gzip 压缩之后，精简和混淆之间的差距将会减小，这些示例也演示了这一点。精简脚本可以降低响应时间，但不会带来混淆的风险。

锦上添花
Icing on the Cake

还有很多其他途径能够减少你的 JavaScript 中浪费的时间。

内联脚本
Inline Scripts

之前的讨论一直在关注外部 JavaScript 文件。内联的 JavaScript 块也应该精简，尽管对于现今的网站这一实践的效果并不很明显。表 10-4 表明尽管十大网站中有 4 个精简了其外部脚本，但只有 3 个精简了其内联脚本。

表 10-4：十大网站的内联脚本精简实践

网站	精简了外部脚本？	精简了内联脚本？
http://www.amazon.com	否	否
http://www.aol.com	否	否
http://www.cnn.com	否	否
http://www.ebay.com	是	否
http://froogle.google.com	是	是
http://www.msn.com	是	是
http://www.myspace.com	否	否
http://www.wikipedia.org	否	否
http://www.yahoo.com	是	是
http://www.youtube.com	否	否

www.TopSage.com

实际上，精简内联脚本比精简外部脚本要容易。不管你使用什么页面生成平台（PHP、Python、Perl CGI 等），JSMin 都会有一个版本能够集成进去。一旦提供了该功能，所有的内联 JavaScript 都会在呈现为 HTML 文档之前被精简。

TopSage.com

压缩和精简

Gzip and Minification

规则 4 强调了对内容进行压缩的重要性，并建议使用 gzip 来完成压缩，这通常可以使大小减小 70%。gzip 压缩比精简更能减小文件的大小——这就是为什么它是规则 4 而这是规则 10。我曾经听到有人问过，如果已经启用了 gzip 压缩，是否还值得进行精简。

表 10-5 和表 10-2 很像，只不过响应是经过压缩的。经过压缩后，JavaScript 有效载荷的平均大小从 85KB（参见表 10-2）降低到了 23KB（参见表 10-5），减少了 73%。这再次印证了规则 4 对这 6 个网站依然是有效的。表 10-5 表明在压缩的基础上使用精简，比起只使用压缩能够平均减少有效载荷 4KB（20%）。有趣的是，混淆并压缩的效率和精简并压缩几乎一样，这再次论证了使用精简可以避免混淆带来的额外风险。

表 10-5：经过 gzip 压缩后 JSMin 和 Dojo Compressor 减小的大小

网站	原始文件压缩后的大小	JSMin 经过压缩后的节省	Dojo Compressor 经过压缩后的节省
http://www.amazon.com/	48KB	7KB（16%）	6KB（13%）
http://www.aol.com/	16KB	1KB（8%）	1KB（8%）
http://www.cnn.com/	29KB	6KB（19%）	6KB（20%）
http://www.myspace.com/	23KB	4KB（19%）	4KB（19%）
http://www.wikipedia.org/	13KB	5KB（37%）	5KB（39%）
http://www.youtube.com/	10KB	2KB（19%）	2KB（20%）
平均	23KB	4KB（20%）	4KB（20%）

总体来说，需要比较的主要数字有——

- 85KB——不进行 JSMin 和 gzip 压缩的 JavaScript 大小
- 68KB——只进行 JSMin 的 JavaScript 大小（节省了 21%）
- 23KB——只进行 gzip 压缩的 JavaScript 大小（节省了 73%）
- 19KB——进行 JSMin 和 gzip 压缩的 JavaScript 大小（节省了 78%）

www.TopSage.com

gzip 压缩产生的影响最大，但精简能够进一步减小文件大小。随着 JavaScript 的使用量和大小不断增长，精简 JavaScript 代码能够得到更多的节省。

大家网 TopSage.com

精简 CSS
Minifying CSS

精简 CSS 能够带来的节省通常要小于精简 JavaScript，因为通常 CSS 中的注释和空白比 JavaScript 少。最大的潜在节省来自于优化 CSS——合并相同的类、移除不使用的类等。CSS 的依赖顺序的本质（这也是将其称为层叠（*Cascading*）样式表的原因）决定了这将是一个复杂的问题。这个领域还需要进一步研究和工具开发。最佳的解决方案还是移除注释和空白，并进行一些直观的优化，比如使用缩写（用“#606”代替“#660066”）和移除不必要的字符串（用“0”代替“0px”）。

对 JavaScript 源代码进行精简

www.TopSage.com

第 11 章

规则 11——避免重定向

Rule 11: Avoid Redirects

大家网 TopSage.com

重定向（*Redirect*）用于将用户从一个 URL 重新路由到另一个 URL。重定向有很多种——301 和 302 是最常用的两种。通常针对 HTML 文档进行重定向，但通常也可能用在请求页面中的组件（图片、脚本等）时。实现重定向可能有很多不同的原因，包括网站重新设计、跟踪流量、记录广告点击和建立易于记忆的 URL。我们将在本章中讨论所有这些方面，但主要要记得的是重定向会使你的页面变慢。

重定向的类型

Types of Redirects

当 Web 服务器向浏览器返回一个重定向时，响应中就会拥有一个范围在 3xx 的状态码。这表示用户代理必需执行进一步操作才能完成请求。以下是几种 3xx 状态码——

- 300 Multiple Choices（基于 Content-Type）
- 301 Moved Permanently
- 302 Moved Temporarily（亦称 Found）
- 303 See Other（对 302 的说明）
- 304 Not Modified
- 305 Use Proxy
- 306（不再使用）
- 307 Temporary Redirect（对 302 的说明）

“304 Not Modified”并不真的是重定向——它用来响应条件 GET 请求，避免下载已经存在于浏览器缓存中的数据，这曾在绪言 B 中介绍过。状态码 306 已经被废弃。

www.TopSage.com

状态码 301 和 302 是使用得最多的。状态码 303 和 307 是在 HTTP 1.1 规范中添加的，用来澄清对 302 的使用（滥用），但是几乎没人采用 303 和 307，绝大多数网站仍然在沿用 302。下面是 301 响应头的一个示例。

```
HTTP 1.1 301 Moved Permanently
Location: http://stevesouders.com/newuri
Content-Type: text/html
```

浏览器会自动将用户带到 `Location` 字段所给出的 URL。重定向所必需的所有信息都出现在这个头中了。响应体通常是空的。不管叫什么名字，301 和 302 响应在实际中都不会被缓存，除非有附加的头——如 `Expires` 或 `Cache-Control` 等——要求它这么做。

还有其他方法可以自动将用户重定向到其他 URL。HTML 文档的头中包含的 `meta refresh` 标签可以在其 `content` 属性所指定的秒数之后重定向用户——

```
<meta http-equiv="refresh" content="0; url=http://stevesouders.com/newuri">
```

JavaScript 也可以用于执行重定向，将 `document.location` 设置为期望的 URL 即可。如果你必须进行重定向，最好的技术是使用标准的 3xx HTTP 状态码，这主要是为了确保后退按钮能够正确工作。与此相关的更多信息，请参见 W3C 文章“Use standard redirects: don't break the back button!”，位于 *http://www.w3.org/QA/Tips/reback*。

重定向是如何损伤性能的

How Redirects Hurt Performance

图 11-1 展示了重定向是如何使用户体验变慢的。第一个 HTTP 请求就是重定向。在重定向完毕并且 HTML 文档下载完毕之前，没有任何东西显示给用户。

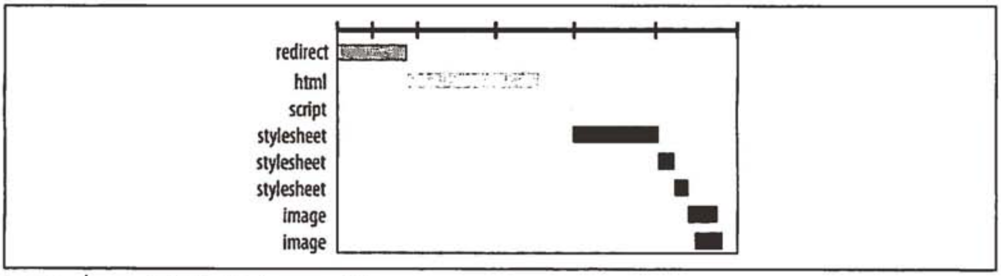


图 11-1：重定向使 Web 页面变慢

TopSage.com
www.TopSage.com

在第 5 章中，我谈到了加快下载样式表的重要性，否则页面的呈现会被阻塞。类似地，第 6 章解释了为什么外部脚本会阻塞页面呈现并阻止并行下载。重定向引起的延迟也很严重，因为它延迟了整个 HTML 文档的传输。在 HTML 文档到达之前，页面中不会呈现出任何东西，也没有任何组件会被下载。在用户和 HTML 文档之间插入重定向延迟了页面中的所有东西。

重定向通常伴随着对 HTML 文档的请求一起使用，但偶尔也会看到将它们用于页面中的组件。图 11-2 展示了 Google Toolbar 产生的 HTTP 请求，它包含 4 个重定向。

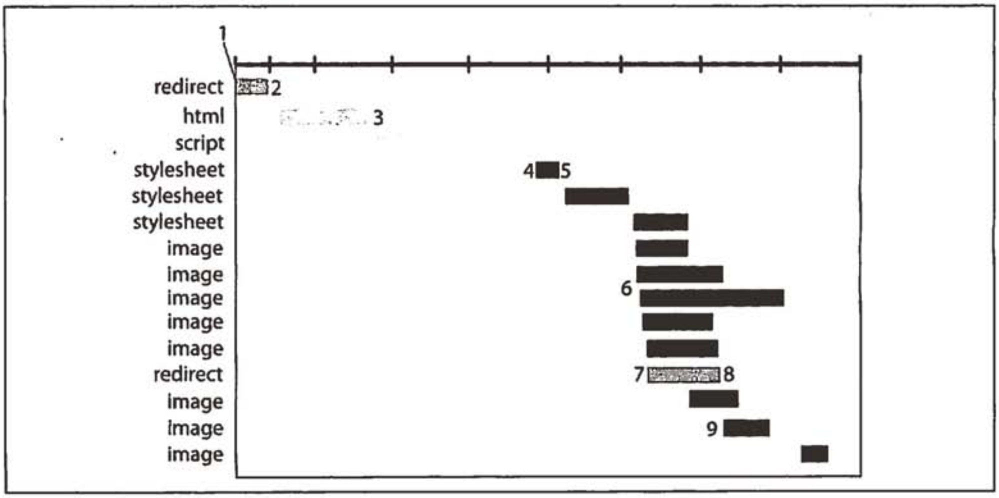


图 11-2：多次重定向，包括一个用于图片的重定向

请求和重定向的顺序非常复杂，因此我将逐个进行介绍——

1. 请求初始 URL *http://toolbar.google.com*。

2. 接收到 302 重定向，其 `Location` 为 *http://toolbar.google.com/T4*。

3. 请求 *http://toolbar.google.com/T4/*。

4. 该 HTML 文档使用 JavaScript 将用户重定向到 *http://www.google.com/tools/firefox/toolbar/index.html*。

5. JavaScript 重定向产生了一个 302 响应，其 `Location` 为 *http://www.google.com/tools/firefox/toolbar/FT3/intl/en/index.html*。此时总共已经进行了三次重定向——一个通过 JavaScript 进行，两个使用 302 状态码进行。

6. 下载 6 个图片。

7. 请求位于 *http://toolbar.google.com/T3/intl/search.gif* 的第 7 个图片。

www.TopSage.com

大家网 TopSage.com

8. 对第 7 个图片的请求产生了一个 302 响应，其 Location 为 *http://toolbar.google.com/intl/search.gif*。
9. 请求最后的 3 个图片。

前 4 个 HTTP 请求（重定向、HTML、脚本和重定向）用于将用户带到期望的 HTML 文档。这些重定向花费了一多半的最终用户响应时间。

重定向使用得很频繁。表 11-1 表明，十大美国网站中有 7 个使用了重定向——2 个在初始页使用，5 个当用户导航到后续页时使用。Google Toolbar 页面中的一个或多个重定向也许是可以避免的。我们来看一些重定向的典型应用以及不会对最终用户响应时间产生如此负面影响的其他选择。

表 11-1：重定向在十大网站的使用情况

网站	使用了重定向
http://www.amazon.com	否
http://www.aol.com	是——后续页
http://www.cnn.com	是——初始页
http://www.ebay.com	是——后续页
http://www.google.com	否
http://www.msn.com	是——初始页
http://www.myspace.com	是——后续页
http://www.wikipedia.org	是——后续页
http://www.yahoo.com	是——后续页
http://www.youtube.com	否

重定向之外的其他选择
Alternatives to Redirects

重定向是解决很多问题的简单方式，但最好使用其他不会减慢页面加载速度的解决方案。下面几节讨论了一些使用重定向的典型解决方案，以及对用户来说更好的替代方案。

缺少结尾的斜线
Missing Trailing Slash

有一种重定向最为浪费、发生得也很频繁，但 Web 开发人员通常都没有意识到它。它发生在 URL 的结尾必须出现斜线（/）而没有出现时。例如，图 11-1 所展示的重定向是在访问 *http://astrology.yahoo.com/astrology* 时产生的。这个请求导致了一个 301 响应，其中包含了一个到 *http://astrology.yahoo.com/astrology/*的重定向。唯一的区别就是在结尾添加了斜线。

www.TopSage.com

大家网 TopSage.com

当缺少结尾的斜线时发送重定向有着很充分的理由——它允许自动索引（autoindexing，自动转到默认的 index.html 上）并且能够获得与当前目录相关的 URL（例如 *logo.gif*）。然而，很多流行的 Web 页面并不依赖自动索引，而是依赖特定的 URL 和处理器。另外，URL 通常也与根目录相关而不是和当前目录相关。

注意当主机名后缺少结尾斜线时不会发生重定向。例如，*http://www.yahoo.com* 不会产生重定向。然而，你在浏览器中看到的最终的 URL 是包含结尾斜线的——*http://www.yahoo.com/*。导致自动产生结尾斜线的原因是，浏览器在进行 GET 请求时必需指定一些路径。如果没有路径，例如 *http://www.yahoo.com*，它就会简单地使用文档根（/）——

```
GET / HTTP 1.1
```

当缺少结尾斜线时发送重定向是很多 Web 服务器的默认行为，包括 Apache。`Alias` 指令是一种简单的方法。另一种选择是使用 `mod_rewrite` 模块，但 `Alias` 更为简单。（注 1）Astrology 网站的问题可以通过向 Apache 配置中添加下列内容来解决——

```
Alias /astrology /usr/local/apache/htdocs/astrology/astrology.html
```

如果使用 Apache 2.0 中的处理器，一种更为清晰的解决方案是使用 `DirectorySlash` 指令（更多信息请访问 *http://httpd.apache.org/docs/2.0/mod/mod_dir.html*）。假设有一个名为 `astrologyhandler` 的处理器，可以像下面这样使用 `DirectorySlash`——

```
<Location /astrology>
  DirectorySlash Off
  SetHandler astrologyhandler
</Location>
```

这些方法都不能解决查找与当前目录相关的 URL 问题，因此页面中组件的 URL 必须与根目录相关。而且，你还必须知道各模块运行的顺序（尤其是 `mod_dir` 和 `mod_autoindex`），因为这样使用 `DirectorySlash` 可能会有安全隐患。

总之，如果你的网站包含目录并使用了自动索引，用户就必须忍受一个到达预期页面的重定向。检查一下你的 Web 日志就能看到发出了多少 301 状态码，这能帮助你认识到多么值得去解决缺少结尾斜线的问题。

注 1：有关 Apache 的 `mod_rewrite` 模块的更多信息，请访问 *http://httpd.apache.org/docs/1.3/mod/mod_rewrite.html*。

www.TopSage.com

大家网 TopSage.com

连接网站

Connecting Web Sites

想象一下网站后端被重写的情形。这经常发生，新的实现中的 URL 很可能会有所不同。将用户从旧的 URL 转移到新的 URL 的最简单的方式就是重定向。重定向是使用定义良好的 API——URL 来整合两个代码基础的一种方式。

将旧网站连接到新网站只是重定向这种常见应用中的表现形式之一。其他形式还包括将一个网站的不同部分连接起来，以及基于一些条件（浏览器类型、用户账户类型等）来引导用户。使用重定向来连接两个网站很简单而且只需要很少的额外代码。

Google Toolbar 页面加载时所需的多个重定向（如图 11-2 所示）就是出于这个目的——连接网站。其后端网站有很多部分（T4、*firefox* 和 FT3）。在后端组件的新版本（如 T5 和 FT4）发布后，可以通过简单地更新重定向来将它们连接到主站点上。

虽然重定向降低了开发的复杂性，但是这也损害了用户体验，正如前面“重定向是如何损伤性能的”一节所介绍的那样。整合两个后端还有其他的选择，但比起简单的重定向需要更多的开发工作，不过这样非但不会损害用户体验，还能使之改善。

- `Alias`、`mod_rewrite` 和 `DirectorySlash`（前面在“缺少结尾的斜线”一节中介绍过）要求除 URL 之外还要提交到一个接口（处理器或文件名），但易于实现。
- 如果两个后端位于同一台服务器上，则它们的代码很可能自己就能够连接。例如，旧的处理器代码可以通过编程调用新的处理器代码。
- 如果域名变了，可以使用一个 CNAME（一条 DNS 记录，用于创建从一个域名指向另一个域名的别名）让两个主机名指向相同的服务器。如果能做到这一点，这里提到的技术（`Alias`、`mod_rewrite`、`DirectorySlash` 和直接连接代码）就是可行的。

跟踪内部流量

Tracking Internal Traffic

重定向经常用于跟踪用户流量的流向。这可以在 Yahoo!的主页（*http://www.yahoo.com*）上看到，这里的很多导航链接都被重定向包装了。例如，Sports 链接的 URL 是 *http://www.yahoo.com/r/26*。单击这个链接将产生 301 响应，其 `Location` 被设置为 *http://sports.yahoo.com/*。通过分析来自 *www.yahoo.com* 的 Web 服务器日志可以得知人们离开 Yahoo!的首页后的流量去向。在这种情况下，离开主页去往 Yahoo! Sports 的人数和日志中 `/r/26` 入口的数量相等。

www.TopSage.com

另一种选择是使用 *Referer*（注 2）日志来跟踪流量去向。每个 HTTP 请求都包含一个 URL，表明从哪个页面发起的请求，也就是引用方（有的时候没有引用页，如当用户键入 URL 或使用书签时）。在这里，当用户从 Yahoo!主页导航到 Sports 页时，*sports.yahoo.com* 的访问日志中会包含一个 Referer，其值为 *http://www.yahoo.com/*。使用 Referer 日志避免了向用户发送重定向，也就改善了响应时间。然而，使用该方法的难处在于，要想对离开 Yahoo! 主页的人进行统计，Yahoo!就必须分析所有目标网站（Sports、Mail、Calendar、Movies，等等）的日志。

对于内部流量——也就是同一家公司的各个网站之间的流量——值得通过建立 Referer 日志来避免重定向，以此节省最终用户响应时间。如果目标网站属于其他公司，就不可能分析其 Referer 日志了。这种情况将在下一节中讨论。

跟踪出站流量
Tracking Outbound Traffic

当你尝试跟踪用户流量时，你会发现链接可能将用户带离你的网站。在这种情况下，使用 Referer 就不太现实了。

这就是 Yahoo! Search 所面临的问题。Yahoo!通过将每个搜索结果链接包装到一个重定向中来解决跟踪的问题。搜索结果的 URL 都指向 *rds.yahoo.com* 并将最终的目标当作参数包含在该 URL 中。例如，下面是指向 Wikipedia 的“Performance”词条的搜索结果链接——

```
http://rds.yahoo.com/[...]5742/**http%3a//en.wikipedia.org/wiki/Performance
```

单击这个搜索结果会访问 *rds.yahoo.com*，它将返回一个 302 响应，其 `Location` 被设置为 *http://en.wikipedia.org/wiki/Performance*。管理员通过分析 *rds.yahoo.com* 上的 Web 服务器日志中的**参数就能跟踪用户去了哪里。这个重定向使得获取目标页面变慢了，对用户体验有负面影响。

跟踪出站流量除重定向之外的选择是使用信标（*beacon*）——一个 HTTP 请求，其 URL 中包含有跟踪信息。跟踪信息可以从信标 Web 服务器的访问日志中提取出来。信标响应通常是一个 1px × 1px 的透明图片；不过 204 响应更为优秀，因为它更小、从来不会被缓存，而且绝对不会改变浏览器的状态。

在 Yahoo! Search 的例子中，目标是无论何时用户单击搜索结果链接时都要发送一个信标。这通过为每个连接提供 `onclick` 处理器来完成（当启用了 JavaScript 时）。`onclick` 处理

注 2：错误的拼写“referrer”应用得非常广泛，以至于已经成为了 HTTP 规范的一部分。

www.TopSage.com

器将调用一个函数，请求一个图片，并在图片的 URL 中包含要跟踪的信息（也就是单击的链接）——

```
<a href="http://en.wikipedia.org/wiki/Performance"
  onclick="resultBeacon(this)">Performance - Wikipedia</a>

<script>
var beacon;
function resultBeacon(anchor) {
    beacon = new Image();
    beacon.src = "http://rds.yahoo.com/?url=" + escape(anchor.href);
}
</script>
```

注意——信标有大量细微之处会带来实现时的可靠性挑战。在前面这种情况下，挑战就是发送信标和页面自身被卸载之间的竞态情形。图片信标的 onload 处理器可以用于确保在卸载文档之前信标应经传送完毕——

```
<a href="http://en.wikipedia.org/wiki/Performance"
  onclick="resultBeacon(this); return false;">Performance - Wikipedia</a>

<script>
var beacon;
function resultBeacon(anchor) {
    beacon = new Image();
    beacon.onload = gotoUrl;
    beacon.onerror = gotoUrl;
    beacon.src = "http://rds.yahoo.com/?url=" + escape(anchor.href);
}

function gotoUrl() {
    document.location = beacon.src;
}
</script>
```

这种方法可能会和使用重定向一样慢，因为两种技巧都必需一个额外的 HTTP 请求。另一种方式是使用 XMLHttpRequest 来发送信标，但在卸载页面之前只需等待请求到达 readyState 2（已发送）即可。这比等待重定向的整个 HTTP 响应要快，但你必须决定是否有必要采取这么复杂的方式。有关使用 XMLHttpRequest 的更多信息，请访问 *http://www. w3.org/TR/XMLHttpRequest*。由于代码示例过于复杂，此处就不再罗列了，但你可以在"XMLHttpRequest 信标"示例中看到它的用法。这里还有一个典型的图片信标示例。

XMLHttpRequest 信标的示例

http://stevesouders.com/hpws/xhr-beacon.php

图片信标的示例

http://stevesouders.com/hpws/redir-beacon.php

www.TopSage.com

虽然这些方式对于很多链接来说过于复杂，但对于使用了 target 属性的链接而言，它们也能很好地工作——

```
<a href="http://en.wikipedia.org/wiki/Performance"
  onclick="resultBeacon(this)"
  target="_blank">Performance - Wikipedia</a>
```

在这种情况下不会出现竞态情形，简单的图片信标就能很好地工作。例如在跟踪弹出式广告的点击（单击）量时，这种方式就很好。单击弹出式广告不会卸载当前文档，图片信标请求能够顺利完成而不会被中断。

美化 URL
Prettier URLs

使用重定向的另一种动机是使 URL 更加美观并且易于记忆。在前面的“缺少结尾的斜线”一节中，我解释了 *http://astrology.yahoo.com/astrology* 是如何将用户重定向到 *http://astrology.yahoo.com/astrology/*（相同的 URL，只是结尾多了个“/”）的。重定向将更多的用户从 *http://astrology.yahoo.com* 带到 *http://astrology.yahoo.com/astrology/*。很明显，*http://astrology.yahoo.com* 更美观、更易于记忆，因此，对于用户来说这种简短的 URL 更好。

“重定向是如何损伤性能的”一节中介绍的 Google Toolbar 重定向是使用重定向来支持美观且易于记忆的 URL 的另外一个例子。想象一下，要想键入或是记住 *http://www.google.com/tools/firefox/toolbar/FT3/intl/en/index.html* 多么困难。还是 *http://toolbar.google.com* 更易于记忆。

关键是要找出一种方式，无需重定向就能拥有如此简洁的 URL。与其让用户忍受额外的 HTTP 请求，最好还是使用“连接网站”一节中介绍的 Alias、mod_rewrite、DirectorySlash 和直接链接代码来避免重定向。

寻找一种避免重定向的方法

www.TopSage.com

规则 12——移除重复脚本
Rule 12: Remove Duplicate Scripts

在一个页面中两次包含同一个 JavaScript 文件会损伤性能。这个错误并不像你想象的那么少见。对美国十大网站的检查表明，其中有两家（CNN 和 YouTube）包含重复脚本。

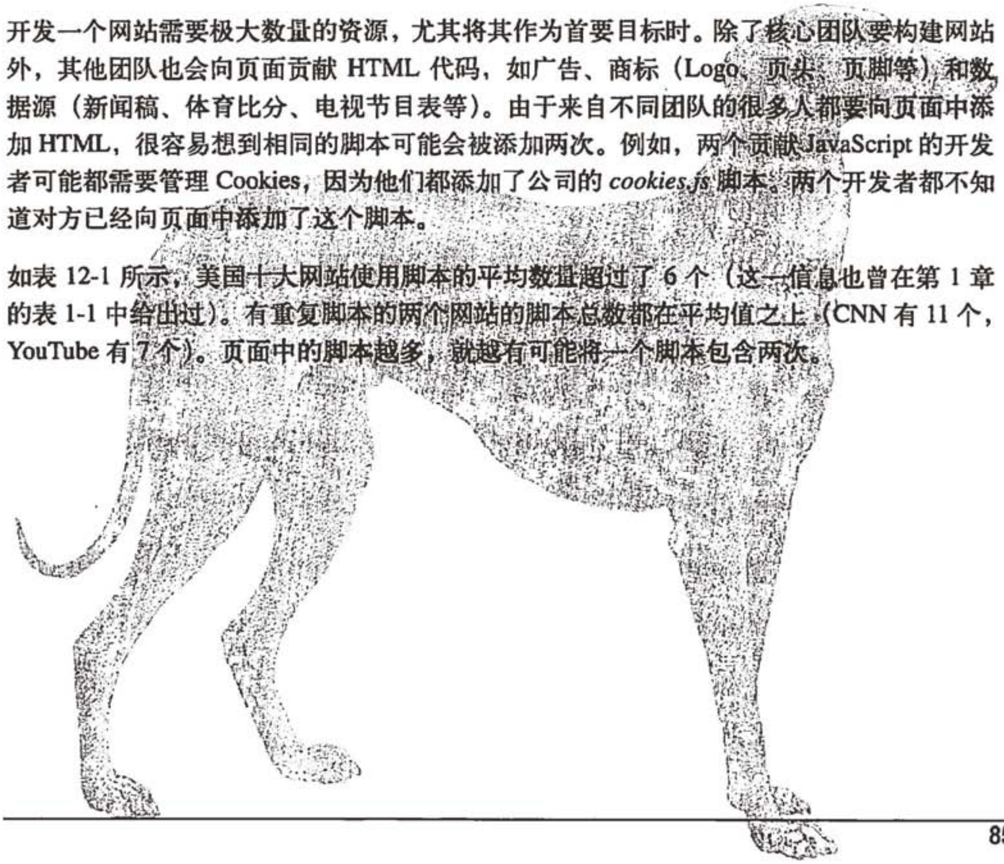
它们是如何产生的？对性能有哪些影响？如何避免？下面我们就来看一看。

重复脚本——确有其事
Duplicate Scripts——They Happen

导致一个脚本的重复有两个主要因素——团队大小和脚本数量。

开发一个网站需要极大数量的资源，尤其将其作为首要目标时。除了核心团队要构建网站外，其他团队也会向页面贡献 HTML 代码，如广告、商标（Logo、页头、页脚等）和数据源（新闻稿、体育比分、电视节目表等）。由于来自不同团队的很多人都要向页面中添加 HTML，很容易想到相同的脚本可能会被添加两次。例如，两个贡献 JavaScript 的开发者可能都需要管理 Cookies，因为他们都添加了公司的 *cookies.js* 脚本。两个开发者都不知道对方已经向页面中添加了这个脚本。

如表 12-1 所示，美国十大网站使用脚本的平均数量超过了 6 个（这一信息也曾在第 1 章的表 1-1 中给出过）。有重复脚本的两个网站的脚本总数都在平均值之上（CNN 有 11 个，YouTube 有 7 个）。页面中的脚本越多，就越有可能将一个脚本包含两次。



www.TopSage.com

表 12-1：十大网站的脚本和样式表数量

网站	脚本	样式表
http://www.amazon.com	3	1
http://www.aol.com	18	1
http://www.cnn.com	11	2
http://www.ebay.com	7	2
http://froogle.google.com	1	1
http://www.msn.com	9	1
http://www.myspace.com	2	2
http://www.wikipedia.org	3	1
http://www.yahoo.com	4	1
http://www.youtube.com	7	3

重复脚本损伤性能
Duplicate Scripts Hurt Performance

重复脚本损伤性能的方式有两种——不必要的HTTP请求和执行JavaScript所浪费的时间。

不必要的 HTTP 请求会发生在 Internet Explorer 中，而不会发生在 Firefox 中。在 Internet Explorer 中，如果脚本被包含两次并且没有被缓存，浏览器会在页面加载期间产生两个 HTTP 请求。“重复脚本——无缓存”示例展示了这一点。

重复脚本——无缓存的示例
http://stevesouders.com/hpws/dupe-scripts.php

对于遵从第 3 章给出的建议并为脚本添加一个长久的 `Expires` 头的人来说，这并不是什么问题，但如果不这样做，当出现两次包含同一个脚本的错误时，用户就不得不忍受一个额外的 HTTP 请求。第 6 章介绍了下载脚本对响应时间有着显著的负面影响。让用户忍受为脚本生成的额外的 HTTP 请求，会让这种负面影响加倍。

即便脚本可以缓存，当用户重新加载页面时也会产生额外的 HTTP 请求。下面的示例包含了可以缓存的脚本。

重复脚本——有缓存的示列
http://stevesouders.com/hpws/dupe-scripts-cached.php

加载一次该页面可以填充缓存，然后单击“Example 2 - Duplicate Scripts - Cached”链接来重新加载它。由于脚本已被缓存，不会为脚本生成 HTTP 请求，但如果单击浏览器的刷新按钮，会产生两个 HTTP 请求。尤其是产生了两个条件 GET 请求。更多信息，请参见绪言 B 中的“条件 GET 请求”一节。

www.TopSage.com

在 Internet Explorer 中除了会产生不必要的 HTTP 请求外，对脚本进行多次求值也会浪费时间。对 JavaScript 进行的多余的执行在 Internet Explorer 和 Firefox 中都存在，与脚本被缓存与否无关。在前面的例子中，重复的脚本会增加显示到页面上的计数器的值。由于脚本被包含两次又被执行两次，所以计数器的值为 2。

每向页面中添加脚本的额外实例一次，多余的下载和执行问题就会出现一次。在下面的例子中，同一个脚本被包含了 10 次，导致被求值 10 次。当你重新加载页面时，会产生 10 个 HTTP 请求（仅在 Internet Explorer 中）。

重复脚本——10 次缓存的示例
http://stevesouders.com/hpws/dupe-scripts-cached10.php

总体来说——

- 在页面中多次包含相同的脚本会使页面变慢。
- 在 Internet Explorer 中，如果脚本没有被缓存，或在重新加载页面时，会产生额外的 HTTP 请求。
- 在 Firefox 和 Internet Explorer 中，脚本会被多次求值。

避免重复脚本
Avoiding Duplicate Scripts

避免意外包含同一个脚本两次的一种方法是，在你的模板系统中实现一个脚本管理模块。包含脚本的典型方式是在 HTML 页面中使用 `SCRIPT` 标签——

```
<script type="text/javascript" src="menu_1.0.17.js"></script>
```

另一种选择是在 PHP 中创建一个称作 `insertScript` 的函数——

```
<?php insertScript("menu.js") ?>
```

当我们处理重复脚本问题时，需要添加处理脚本的依赖性和版本的功能。`insertScript` 的一个简单的实现如下所示——

```
<?php
function insertScript($jsfile) {
    if ( alreadyInserted($jsfile) ) {
        return;
    }
    pushInserted($jsfile);

    if ( hasDependencies($jsfile) ) {
        $dependencies = getDependencies($jsfile);
        Foreach ($dependencies as $script) {
            insertScript($script);
        }
    }
```


www.TopSage.com

```
        echo '<script type="text/javascript" src="' . getVersion($jsfile) . '"></script>
    ";
    }
    ?>
```

第一次插入脚本时，可以到达 pushInserted。这会把脚本添加到页面的 alreadyInserted 列表中。当意外地再次插入该脚本时，将会对 alreadyInserted 进行检测，因而不会再次插入脚本，也就解决了重复脚本的问题。

如果该脚本依赖其他脚本，则其所依赖的脚本也将被插入。在这个示例中，*menu.js* 依赖于 *event.js* 和 *utils.js*。你可以使用哈希表或数据库来搜集这些依赖关系。对于简单的网站，可以手动地维护依赖关系。对于复杂的网站，可以选择通过扫描脚本查找符号定义来自动生成依赖关系。

最后，脚本被传送到页面中。这里的一个关键函数是 getVersion。这个函数会查找脚本（这里是 *menu.js*）并返回追加了对应的版本号后的文件名（即 *menu_1.0.17.js*）。在第 3 章中，我介绍了在使用了长久的 Expires 头之后，一旦文件内容发生变化，文件名也必须改变（参见第 3 章中“修订文件名”一节）；这可以通过向组件的文件名中添加版本号来实现。该脚本管理模块的另一个好处是将上述功能集中在了 insertScript 函数中。一旦脚本发生变化，只要简单地更新 getVersion 的代码，所有的页面就可以开始使用新的文件名了。无需修改任何 PHP 模板就能让页面开始使用新的版本。

确保脚本只被包含一次

www.TopSage.com

规则 13——配置 ETag
Rule 13: Configure ETags

减少呈现页面时所必需的 HTTP 请求的数量是加速用户体验的最佳方式。可以通过最大化浏览器缓存组件的能力来实现这一目标，但当网站被宿主在多于一台服务器上时，`ETag` 头可能会阻碍缓存。在这一章里，我将介绍 ETag 是什么以及其默认实现是如何使 Web 页面变得缓慢的。

ETag 是什么？
What's an ETag?

实体标签（*Entity Tag*，*ETag*）是 Web 服务器和浏览器用于确认缓存组件的有效性的一种机制。在进入 ETag 的细节之前，我们先回顾一下组件是如何被缓存和确认的。

Expires 头
Expires Header

浏览器下载组件时，会将它们存储到缓存中。在后续的页面查看中，如果缓存的组件是“新鲜的”，浏览器就会从磁盘上读取它，避免产生 HTTP 请求。如果组件没有过期，那么它就是新鲜的，这取决于 `Expires` 头的值。我们来看一个例子。

当请求一个组件时，服务器会根据其选项在响应中返回一个 `Expires` 头——

```
Expires: Thu, 15 Apr 2010 20:00:00 GMT
```

第 3 章建议设置一个很久之后的过期日期。多久称之为“久”取决于正在谈论的组件。一个广告图片可能每天都会过期，而公司 Logo 可能 10 年后才过期。HTTP 规范（*http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.21*）建议服务器不应将 `Expires` 日期设置为未来一年之后，但这只是一个指导原则；浏览器支持超过未来一年的 `Expires` 过期日期。对于不大会过期的组件，将过期日期设置得久远一些能够更有效地避免 HTTP 请求。

www.TopSage.com

大家网 TopSage.com

条件 GET 请求
Conditional GET Requests

如果缓存的组件过期了（或者用户明确地重新加载了页面），浏览器在重用它之前必须首先检查它是否仍然有效。这称作一个**条件 GET 请求**（参见绪言 B 中的“条件 GET 请求”一节）。不幸的是浏览器必须产生这个 HTTP 请求，执行有效性检查，但这仍比简单地下载所有已过期的组件效率要高。如果浏览器缓存中的组件是有效的（即它能够和原始服务器上的组件相匹配），原始服务器不会返回整个组件，而是返回一个“304 Not Modified”状态码。

服务器在检测缓存的组件是否和原始服务器上的组件匹配时有两种方式——

- 比较最新修改日期
- 比较实体标签

最新修改日期
Last-Modefied Date

原始服务器通过 `Last-Modified` 响应头来返回组件的最新修改日期。

```
GET /i/yahoo.gif HTTP 1.1
Host: us.yimg.com
```

```
HTTP 1.1 200 OK
Last-Modified: tue, 12 Dec 2006 03:03:59 GMT
Content-Length: 1195
```

在这个示例中，浏览器缓存了组件（在这里是 Yahoo!的 Logo）以及它的最新修改日期。下一次请求 *http://us.yimg.com/i/yahoo.gif* 时，浏览器会使用 `If-Modified-Since` 头将最新修改日期传回到原始服务器以进行比较。如果原始服务器上组件的最新修改日期与浏览器传回的值匹配，会返回一个 304 响应，而不会传送 1195 字节的数据。

```
GET /i/yahoo.gif HTTP 1.1
Host: us.yimg.com
If-Modified-Since: Tue, 12 Dec 2006 03:03:59 GMT
```

```
HTTP 1.1 304 Not Modified
```


www.TopSage.com

大家网
TopSage.com

实体标签
Entity Tags

ETag 提供了另外一种方式，用于检测浏览器缓存中的组件与原始服务器上的组件是否匹配（“实体”是我们之前提到的“组件”的另一种称呼）。ETag 在 HTTP 1.1 中引入。ETag 是唯一标识了一个组件的一个特定版本的字符串。唯一的格式约束是该字符串必须用引号引起来。原始服务器使用 `ETag` 响应头来指定组件的 ETag。

```
GET /i/yahoo.gif HTTP 1.1
Host: us.yimg.com
```

```
HTTP 1.1 200 OK
Last-Modified: Tue, 12 Dec 2006 03:03:59 GMT
ETag: "10c24bc-4ab-457e1c1f"
Content-Length: 1195
```

ETag 的加入为验证实体提供了比最新修改日期更为灵活的机制。例如，如果实体依据 `User-Agent` 或 `Accept-Language` 头而改变，实体的状态可以反映在 ETag 中。

此后，如果浏览器必须验证一个组件，它会使用 `If-None-Match` 头将 ETag 传回原始服务器。如果 ETag 是匹配的，就会返回 304 状态码，使响应减小了 1195 字节。

```
GET /i/yahoo.gif HTTP 1.1
Host: us.yimg.com
If-Modified-Since: Tue, 12 Dec 2006 03:03:59 GMT
If-None-Match: "10c24bc-4ab-457e1c1f "
```

```
HTTP 1.1 304 Not Modified
```

ETag 带来的问题
The Problem with ETags

ETag 的问题在于，通常使用组件的某些属性来构造它，这些属性对于特定的、寄宿了网站的服务器来说是唯一的。当浏览器从一台服务器上获取了原始组件，之后，又向另外一台不同的服务器发起条件 GET 请求时，ETag 是不会匹配的——而对于使用服务器集群来处理请求的网站来说，这是很常见的一种情况。默认情况下，对于拥有多台服务器的网站，Apache 和 IIS 向 ETag 中嵌入的数据都会大大地降低有效性验证的成功率。

www.TopSage.com

Apache 1.3 和 2.x 的 ETag 格式是 `inode-size-timestamp`。文件系统使用 inode 来存储诸如文件类型、所有者、组和访问模式等信息。尽管在多台服务器上一个给定的文件可能位于相同的目录、具有相同的文件大小、权限、时间戳等，从一台服务器到另一台服务器，其 inode 仍然是不同的。

IIS 5.0 和 6.0 在 ETag 上有着类似的问题。IIS 上 ETag 的格式是 `Filetimestamp:ChangeNumber`。`ChangeNumber` 适用于跟踪 IIS 配置变化的计数器。对于一个网站背后的所有 IIS 服务器来说，`ChangeNumber` 不大可能相同。

最后的结果是，对于完全相同的组件，从一台服务器到另一台，Apache 和 IIS 产生的 ETag 是不会匹配的。如果 ETag 不匹配，用户就不会按照 ETage 的设计计划那样接收到更小更快的 304 响应；相反，它们会收到普通的 200 响应以及组件的所有数据。如果你只在一台服务器上寄宿网站，这不是什么问题，但如果你使用了服务器集群，则组件的下载次数可能会比必须进行下载的次数多得多，这将导致性能的下降。

对组件进行不必要的重新加载还会影响服务器的性能并增加带宽开销。如果你的 Round-Robin Rotation 集群中有 *n* 台服务器，下一次用户缓存中的 ETag 能和服务器匹配的概率是 *1/n*。如果你有 10 台服务器，则用户只有 10%的机会得到正确的 304 响应，其余 90%的机会会得到无用的 200 响应并下载所有的数据。

ETag 还降低了代理缓存的效率。代理后面的用户缓存的 ETag 经常和代理缓存的 ETag 不匹配，这导致不必要的请求被发送到原始服务器。用户和代理之间不会出现 304 响应，而是会产生两个（又慢又大的）200 响应——一个是从原始服务器到代理的，另一个是从代理到用户的。ETag 的默认格式还可能会引入安全性弱点（注 1）。

这可真差劲。

`If-None-Match` 比 `If-Modified-Since` 具有更高的优先级。你可能希望如果 ETag 不匹配但最新修改日期是相同的，也能发送一个“304 Not Modified”响应，但实际并不是这样。依据 HTTP 1.1 规范（*http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.3.4*），如果请求中同时出现了这两个头，则原始服务器“禁止（MUST NOT）返回 304（Not Modified），除非请求中的条件头字段全部一致”。实际上如果根本没有 `If-None-Match` 头反而会更好一些。下一节将讨论解决方法。

注 1：更多信息请参见位于 *http://www3.ca.com/securityadvisor/vulninfo/vuln.aspx?ID=7196* 的 Web 文章“Apache http daemon file inode disclosure vulnerability”。

www.TopSage.com

大家网
TopSage.com

ETag——用还是不用

ETags: use'Em of Lose'Em

如果你在多台服务器上寄宿你的网站，而且你使用的是具有默认 ETag 配置的 Apache 或 IIS，你的用户将面对缓慢的页面、你的服务器会具有很高的负载、你会消耗大量的带宽，而且代理也不能有效地缓存你的内容。“但等一下！”你会说，“我遵守了规则 3 并为我的组件添加了长久的 `Expires` 头。不应该出现条件 GET 请求。”

即便你的组件具有长久的 `Expires` 头，一旦用户单击了 Reload 或 Refresh 按钮，依然会产生条件 GET 请求。没有办法能够绕过它——ETag 带来的问题是必须面对的。

一种选择是对 ETag 进行配置，以利用其灵活的验证能力。例如可以使用一段根据浏览器是否为 Internet Explorer 而变化的脚本。如果使用 PHP 来生成脚本，你可以通过设置 `ETag` 头来反映浏览器状态——

```
<?php
if ( strops($_SERVER["HTTP_USER_AGENT"], "MSIE") ) {
    header("ETag: MSIE");
}
else {
    header("ETag: notMSIE");
}
?>
```

如果你的组件必须通过最新修改日期之外的一些东西来进行验证，则 ETag 是一种强大的方法。

如果你无须自定义 ETag，最好简单地将其移除。Apache 和 IIS 都将 ETag 视为一个性能问题，并建议修改 ETage 的内容（更多细节请参见 *http://www.apacheweek.com/issues/02-01-18*、*http://support.microsoft.com/?id=922733* 和 *http://support.microsoft.com/?id=922703*）。

Apache 1.3.23 版和之后版本支持 `FileETag` 指令。使用这一指令，可以从 ETag 中移除 `inode` 值，只留下大小和时间戳作为组件的 ETag。类似地，在 IIS 中可以为所有服务器设置相同的 `ChangeNumber`，保留文件的时间戳作为 ETag 中仅有的另一块信息。

遵从这一建议之后，ETag 中将只包含大小和时间戳（Apache）或只有时间戳（IIS）。然而，因为这些都是重复信息，所以最好将 ETag 完全移除——`Last-Modified` 头可以提供完全等价的信息，而且移除 ETag 可以减小响应和后续请求的 HTTP 头的大小。这一节中引用

www.TopSage.com

大家网 TopSage.com

的 Microsoft Support 文章中介绍了如何移除 ETag。在 Apache 中，只需向 Apache 配置文件中简单地添加下面一行配置就能移除 ETag——

```
FileETag none
```

现实世界中的 ETag
ETags in the Real World

表 13-1 表明，美国十大网站中有 6 个为其大部分组件使用了 ETag。可以清楚地看到，其中有 3 家修改了 ETag 格式，移除了 inode（Apache）或 ChangeNumber（IIS）。4 家或更多家包含了没有修改过的 ETag，并由此导致了前面讨论过的性能问题。

表 13-1：十大网站的 ETag 观察

网站	带有 ETag 的组件	已修正
http://www.amazon.com	0%（0/24）	n/a
http://www.aol.com	5%（3/63）	是
http://www.cnn.com	83%（157/190）	否
http://www.ebay.com	86%（57/66）	否
http://www.google.com	0%（0/5）	n/a
http://www.msn.com	72%（42/58）	否
http://www.myspace.com	84%（32/38）	是和否
http://www.wikipedia.org	94%（16/17）	未知
http://www.yahoo.com	0%（0/34）	n/a
http://www.youtube.com	70%（43/61）	是

在集群服务器中具有不同 ETag 的组件的一个示例是来自 *http://msn.com* 的 *http://stc.msn.com/br/hp/en-us/css/15/blu.css*。第一次请求该示例时的 HTTP 头中包含一个值为 80b31d5a4776c71:6e0 的 ETag。

```
GET /br/hp/en-us/css/15/blu.css HTTP 1.1
Host: stc.msn.com
```

```
HTTP 1.1 200 OK
Last-Modified: Tue, 03 Apr 2007 23:25:23 GMT
ETag: "80b31d5a4776c71:6e0"
Content-Length: 647
Server: Microsoft-IIS/6.0
```

第一次重新加载时，ETag 是匹配的，并发送了 304 响应。Content-Length 头没有出现在响应中，因为 304 状态码告诉浏览器使用其缓存中的内容。

```
GET /br/hp/en-us/css/15/blu.css HTTP 1.1
Host: stc.msn.com
If-Modified-Since: Tue, 03 Apr 2007 23:25:23 GMT
If-None-Match: "80b31d5a4776c71:6e0"
```


www.TopSage.com

大家网 TopSage.com

```
HTTP 1.1 200 OK
ETag: "80b31d5a4776c71:6e0"
Last-Modified: Tue, 03 Apr 2007 23:25:23 GMT
```

第二次重新加载时，ETag 变为 80b31d5a4776c71:47b。这次没有返回更快的 304 响应，而是返回了更大的 200 响应和完整的内容。

```
GET /br/hp/en-us/css/15/blu.css HTTP 1.1
Host: stc.msn.com
If-Modified-Since: Tue, 03 Apr 2007 23:25:23 GMT
If-None-Match: "80b31d5a4776c71:6e0"
```

```
HTTP 1.1 200 OK
Last-Modified: Tue, 03 Apr 2007 23:25:23 GMT
ETag: "80b31d5a4776c71:47b"
Content-Length: 647
Server: Microsoft-IIS/6.0
```

尽管 ETag 改变了，我们依然知道这是同一个组件。其大小（647 字节）是相同的。最新修改日期（03 April 2007 23:25:23）是也相同的。ETag 也几乎是相同的。我们来近距离看一下 ETag 头——

```
ETag: "80b31d5a4776c71:6e0"
ETag: "80b31d5a4776c71:47b"
```

从响应的 Server 头可以确定这是来自 IIS 的。前面介绍过，IIS 的默认 ETag 格式是 Filetimestamp:ChangeNumber。两个 ETag 有着相同的 Filetimestamp 值（80b31d5a4776c71）。这并不奇怪，因为 Last-Modified 头表明两个组件具有相同的修改日期。ChangeNumber 是 ETag 中不同的部分。尽管很令人失望，但这并不奇怪，因为前面引用的 Microsoft Support 文章已经陈述了这一点，这正是引发性能问题的原因。从 ETag 中移除 ChangeNumber 或完全移除 ETag 可以避免当数据已经位于浏览器缓存中时进行不必要的和低效的下载。

配置或移除 ETag

www.TopSage.com

第 14 章

规则 14——使 Ajax 可缓存

Rule 14: Make Ajax Cacheable

大家网 TopSage.com

人们经常问起本书中的规则是否适用于 Web 2.0 应用程序。很明显它们可以。而且，这一章要讨论的规则正是 Yahoo!在开始 Web 2.0 应用程序后遇到的。在这一章中，我将介绍 Web 2.0 是什么、Ajax 是如何适用于 Web 2.0 的，以及你可以针对 Ajax 做出的重要性能改进。

Web 2.0、DHTML 和 Ajax

Web 2.0, DHTML, and Ajax

图 14-1 展示了 Web 2.0、DHTML 和 Ajax 之间的关系。本图并不表示 Ajax 仅用于 DHTML 或 DHTML 仅用于 Web 2.0 应用程序，但它确实表明了 Web 2.0 包含很多概念，其中之一就是 DHTML，Ajax 也是 DHTML 中的一项关键技术。对 Web 2.0 及其包含什么的介绍本身就是一本（或很多本）书了，我们这里只想让读者对这些术语有个一般的了解。下面我会给出一些简短的定义和更多信息的参考资料。

图 14-1：Web 2.0、DHTML 和 Ajax 之间的关系

www.TopSage.com

Web 2.0

O'Reilly Media 在 2004 年第一次举办 Web 2.0 大会时首次使用了术语"Web 2.0"。该术语并不表示下一代 Internet 将会使用用于社会和商业方面的众多技术。它仅暗指了 Internet 社区中丰富的维基、博客和播客所带来的网站。Web 2.0 的关键概念包括类似应用程序的用户界面和来自多个 Web Services 的聚合信息。Web 页面变得越来越像一个具有良好定义的输入、输出的应用程序。DHTML 和 Ajax 是实现这些概念的技术。Tim O'Reilly 在其 Web 文章"Web 2.0 Compact Definition: Try Again"（*http://radar.oreilly.com/archives/2006/12/web_20_compact.html*）中提供了 Web 2.0 的众多参考定义中的一种。

DHTML

动态 HTML 允许在页面加载完毕后，HTML 页面的表现能够变化。这使用 JavaScript 和 CSS 与浏览器的文档对象模型（Document Object Model，DOM）进行交互来实现。例如当用户将鼠标悬浮在链接上面时，页面表现会发生变化，如树状控件可以展开和折叠，页面中的层叠菜单也会变化。更复杂的 DHTML 页面可能会根据用户的意图重绘整个页面，例如，从浏览 Email 的收件箱转到撰写邮件消息时，Ajax 是 DHTML 中使用的一项技术，客户端可以获取和显示用户请求的新信息而无需重新加载页面。

Ajax

术语"Ajax"由 Jesse James Garrett 于 2005 年提出（注 1）。Ajax 表示"异步的 JavaScript 和 XML（Asynchronous JavaScript and XML）"（尽管今天除了 XML 有很多其他选择，最著名的是 JSON）。Ajax 不是一个单独的、需要许可证的技术，而是一组技术，包括 JavaScript、CSS、DOM 和异步数据获取。Ajax 的目的是为了突破 Web 本质的开始-停止交互方式。向用户显示一个白屏然后重绘整个页面不是一种好的用户体验。而 Ajax 在 UI 和 Web 服务器之间插入了一层。这个 Ajax 层位于客户端，与 Web 服务器进行交互以获取请求的信息，并与表现层交互，仅更新那些必要的组件。它将 Web 体验从"浏览页面"转变为"与应用程序进行交互"。

注 1：Jesse James Garrent，"Ajax: A New Approach to Web Applications"，*http://www.adaptivepath.com/publications/essays/archives/000385.php*。

www.TopSage.com

Ajax 背后的技术比这个术语本身包含的要宽泛得多。IFrame——1996 年首次由 Internet Explorer 3 引入——允许在页面中异步加载内容，至今仍有一些 Ajax 应用程序在使用这一技术。XMLHttpRequest——我认为是 Ajax 的心脏——1999 年在 Internet Explorer 中提出（名为 XMLHTTP），2003 年在 Mozilla 中出现。针对 Ajax 的 W3C XMLHttpRequest 规范提议于 2006 年 4 月发布。

我强烈建议使用 Yahoo! UI（YUI） Connection Manager（*http://developer.yahoo.com/yui/connection*）来进行 Ajax 开发。它处理了 XMLHttpRequest 的浏览器兼容性问题，并具有优秀的文档和代码示例。

异步与即时

Asynchronous = Instantaneous?

Ajax 的一个明显的优点就是向用户提供了即时反馈，因为它异步地从后端 Web 服务器请求信息。在前面引用的 Web 文章中，Jesse James Garrett 在提到“所有东西几乎都是即时的”时，并使用 Google Suggest 和 Google Maps 作为了 Web 界面的示例。

小心！使用 Ajax 并不保证用户就不会一边玩弄自己的手指一边等着“异步的 JavaScript 和 XML”返回响应。我讨厌在拨号连接上使用 Google Maps 和 Yahoo! Maps。在很多应用程序中，用户是否需要等待取决于如何使用 Ajax。前端工程师又一次要担当起明确和遵守最佳实践的责任，以确保用户体验能够很快。

用户是否需要等待的关键因素在于 Ajax 请求是被动的还是主动的。**被动请求**（*Passive Request*）是为了将来使用而预先发起的。例如，在一个基于 Web 的 Email 客户端中，可能会使用被动请求在用户真正需要之前下载用户的地址簿。经过被动加载，当用户需要为一个 Email 消息添加地址时，客户端能够确保地址簿已经存在于缓存中。**主动请求**（*Active Request*）是基于用户当前的操作而发起的。例如查找所有与用户的搜索条件相匹配的 Email 消息。

后面的例子展示了，即便主动的 Ajax 请求是异步的，用户可能仍然需要等待响应。不过我们真的还是应该感谢 Ajax，用户不必忍受整个页面的重新加载了，而且当用户等待时，UI 仍然是可响应的。不过，用户在进行进一步操作之前仍可能需要坐在那里，等待搜索结果显示出来。记住“异步”并没有暗示“即时”，这一点很重要。我非常认同 Jesse James Garrett 最后给出的 FAQ。

> 问：Ajax 应用程序总是能够比传统 Web 应用程序提供更好的体验吗？
>
> 答：不一定。Ajax 给交互设计师带来了更多的灵活性。然而，我们的能力越强大，在使用的时候越应该小心。我们必须小心地使用 Ajax，确保它是优化了应用程序的用户体现，而不是恶化了。

www.TopSage.com

我还认同他的另一个评论——

> 会好的。

让我们开心地去探索如何使用 Ajax 优化用户体验，避开可能恶化用户体验的常见缺陷吧。

优化 Ajax 请求

Optimizing Ajax Requests

前面一节明确了当发起主动 Ajax 请求时，用户可能仍须等待。要改善性能，优化这些请求很重要。优化主动 Ajax 请求的技术同样适用于被动 Ajax 请求，但由于主动请求对用户体验的影响更大，所以我们从这里开始。

要找出 Web 应用程序中所有的主动 Ajax 请求，请启动你喜欢的抓包工具。（第 15 章中“如何进行测试”一节提到了我喜欢的抓包工具——IBM Page Detailer）。在 Web 应用程序加载完毕后，开始使用它，同时观察抓包工具中显示出来的 Ajax 请求。这些主动 Ajax 请求就是必须进行优化以实现更佳性能的。

改善这些主动 Ajax 请求的最重要的方式就是使响应可缓存，如第 3 章中的讨论。我们讨论过的其他 13 个规则中有一些也适用于 Ajax 请求——

- 规则 4——压缩组件
- 规则 9——减少 DNS 查找
- 规则 10——精简 JavaScript
- 规则 11——避免重定向
- 规则 13——ETag——用还是不用

然而，规则 3 是最重要的。我建立一个新的规则，仅仅是在一个新的环境中简单地重申前面的规则，这有些不公平。但我发现，由于 Ajax 是如此地崭新和不一般，这些性能改进必须明确地单独提出来。

现实世界中的 Ajax 缓存

Caching Ajax in the Real World

我们来看几个例子，看看现实世界中 Ajax 是如何贯彻这些性能指导方针的。

Yahoo! Mail

在第一个例子中，我们看一下 Yahoo! Mail 的 Ajax 版本（*http://mail.yahoo.com*），在写作本书时它还是 beta 版本。

www.TopSage.com

当用户启动 Yahoo! Mail 的 Ajax 版本时，它会下载前三个 Email 消息的正文。这是一个聪明的主动 Ajax 请求。用户很有可能单击这些 Email 消息中的一个或几个，它们已经被下载到客户端，这意味着用户无需进行任何 Ajax 请求就能看到他的 Email 消息。

如果用户想查看一个不在前三个消息之列的 Email 消息，就会产生一个主动 Ajax 请求。用户在阅读 Email 消息之前必须等待响应。我们来看一下 HTTP 头。

```
GET /ws/mail/v1/formrpc?m=GetMessage[snip...] HTTP 1.1
Host: us.mg0.mail.yahoo.com
Accept-Encoding: gzip,deflate
```

```
HTTP 1.1 200 OK
Date: Mon, 23 Apr 2007 23:22:57 GMT
Cache-Control: no-store, private
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: text/xml; charset=UTF-8
Content-Encoding: gzip
Connection: keep-alive
```

现在假设用户离开了 Yahoo! Mail 并浏览了其他网站。之后，他又回到 Yahoo! Mail 并再次单击第四个 Email 消息。不出所料，又一次发送了完全相同的请求，因为之前的 Ajax 响应并没有保存在浏览器的缓存中。没有被缓存的原因是响应中包含一个值为 `no-store` 的 `Cache-Control` 头，以及一个日期为过去某一天的 `Expires` 头。这些都告诉浏览器不要缓存响应。但是，如果收件箱没有发生变化，两次响应的内容是完全相同的。

如果使用一个长久的 `Expires` 头（参见第 3 章）来替换这些头，响应将被缓存并从磁盘上进行读取，从而得到更快的用户体验。这对于一些开发者看来是违反直觉的——毕竟，这应该是一个动态生成的响应，只包含与这世界上的一个用户相关的信息。缓存这些数据看上去没有任何意义。要记得的很重要的一点是，一个用户可能每天或每周进行很多次请求。如果你可以为他缓存响应，就会看到缓慢的和快速的用户体验之间的差距。

使这些 Ajax 请求可缓存，除了改变 HTTP 头之外还需要进行更多的工作。响应的个性化和动态本质必须被反映到缓存中。可供采用的最好的方式是使用查询字符串参数。例如，这个响应只对当前用户有效。可以将用户名放到查询字符串中来做到这一点——

```
/ws/mail/v1/formrpc?m=GetMessage&yid=steve_souders
```

还有很重要的一点是要反映出确切的消息。我们不能使用`&msg=4`，因为收件箱中的“第 4 个”消息随时会变。应该使用对于该用户的所有消息来说唯一的消息 ID 来解决这个问题——

www.TopSage.com

```
/ws/mail/v1/formrpc?m=GetMessage&yid=steve_souders&msgid=001234
```

该响应可能会因为数据隐私原因而不能缓存。当数据被认为是私有的时，大多会使用 Cache-Control: no-store。在使用了这个头之后，响应根本就不会被写入到磁盘上。但是，HTTP 规范警告说不要依赖这一机制来确保数据的隐私性，恶意的或危险的缓存会完全忽略 Cache-Control: no-store 头。

处理数据隐私性的另外一种更好的方法是使用安全通信协议如安全套接字层（Secure Sockets Layer，SSL）。SSL 响应是可缓存的（在 Firefox 只能用于当前浏览器会话），因此它提供了一种妥协——在确保数据隐私的同时在当前会话中缓存响应以改善用户体验。

遍历其他相关的性能规则，我们可以从这个实例中看到很多积极的性能特征。响应被压缩了（规则 4）。域名同时也用在了页面的其他请求中，这有助于避免额外的 DNS 查找（规则 9）。XML 响应被尽可能地精简了（规则 10）。没有使用重定向（规则 11）。而且 ETag 也被移除了（规则 13）。

Google Spreadsheets

Google Docs & Spreadsheets（*http://docs.google.com*）提供了一个 Ajax 电子表格应用程序，写作本书时它还是 beta 版。

在典型的工作流中，用户会创建一个电子表格并将其保存在文档列表中。我们来检测一下当用户返回并打开电子表格时会发生什么。图 14-2 展示了当打开电子表格时的 HTTP 流量——产生了 10 个主动 Ajax 请求。要交代一下的是，这些 Ajax 请求并不全是使用 XMLHttpRequest 进行的排他性的 XML 获取。在 Google Spreadsheets 中，这些请求有的使用了 XMLHttpRequest，也有的使用了 IFrame。

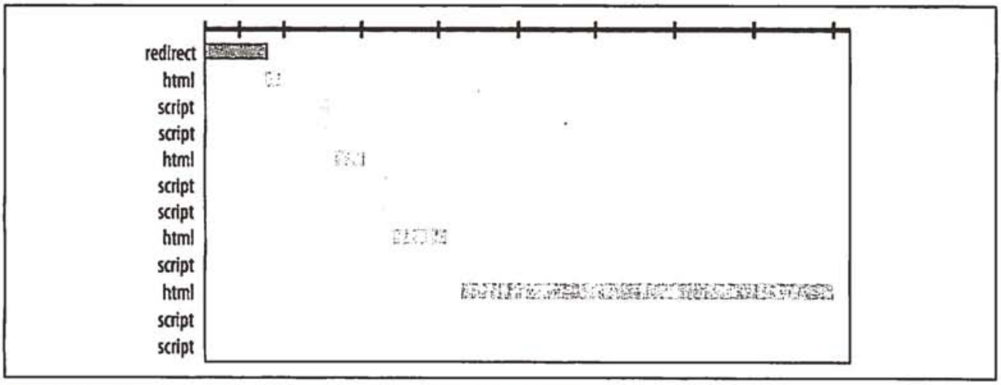


图 14-2：Google Spreadsheets 中的主动 Ajax 请求

www.TopSage.com

如果用户关闭该电子表格并重新打开它，会再次产生这 10 个请求。这是因为这些请求都没有被缓存。这些请求中很多都很小，但其中一个 HTML 请求是 47KB（压缩之前）。我们看一下这个请求的 HTTP 头。

TopSage.com

```
GET /ar?id=[snip...]&srow=0&erow=100 HTTP 1.1
Host: spreadsheets.google.com
Accept-Encoding: gzip,deflate
```

```
HTTP 1.1 200 OK
Content-Type: text/html; charset=UTF-8
Cache-Control: private
Content-Encoding: gzip
Date: Tue, 24 Apr 2007 23:37:13 GMT
```

每次用户打开电子表格时，都会不出所料地产生该 Ajax 请求。响应中没有任何一个头告诉浏览器对其进行缓存。在我的测试中，我没有修改过该电子表格，因此每次打开这个电子表格时该响应都是一样的。实际上，10 个请求中有 8 个都是一样的，这就产生了一个问题，是否应该缓存它们。

和 Yahoo! Mail 示例一样，缓存该电子表格不像添加一个长久的 Expires 头那样简单。如果用户修改了该电子表格，我们必须确保产生变化后不会再使用缓存的请求。简单的解决方法还是使用查询字符串。Google Spreadsheets 的后端应该具有一个时间戳，来表示末次修改发生的时间，并将其嵌入到 Ajax 请求的查询字符串中——

```
/ar?id=[snip...]&srow=0&erow=100&t=1177458941
```

尽管 Ajax 请求不是可缓存的，其他性能指导原则都被成功地实现了。响应被压缩了（规则 4）。和 Google 的很多其他网站一样，域名查找被最小化了（规则 9）。脚本被精简了（规则 10）。没有使用重定向（规则 11），而且 ETag 也被移除了（规则 13）。

确保 Ajax 请求遵守性能指导，尤其应具有长久的 Expires 头

www.TopSage.com

析构十大网站

Deconstructing 10 Top Sites

TopSage.com

下面要做的是使用本书介绍的规则和工具对美国十大网站进行考察。这些分析为指出现实世界中的网站的性能改善给出了示例。我希望在浏览完这些示例之后，你能站在性能提倡者的角度以批判的眼光看待网站。

页面大小、响应时间、YSlow 等级

Page Weight, Response Time, YSlow Grade

表 15-1 展示了 2007 年早期测量的美国十大网站的页面大小、响应时间和 YSlow 等级。YSlow 是 Yahoo!开发的一个性能工具，会根据页面实践本书所介绍的性能规则的优劣程度产生一个单独的得分（A 为最好，F 为最差）。更多信息请参见下一节“如何进行测试”。

表 15-1：美国十大网站的性能摘要

	页面大小	响应时间	YSlow 等级
Amazon	405KB	15.9 秒	D
AOL	182KB	11.5 秒	F
CNN	502KB	22.4 秒	F
eBay	275KB	9.6 秒	C
Google	18KB	1.7 秒	A
MSN	221KB	9.3 秒	F
MySpace	205KB	7.8 秒	D
Wikipedia	106KB	6.2 秒	C
Yahoo!	178KB	5.9 秒	A
YouTube	139KB	9.6 秒	D

www.TopSage.com

不出所料，页面大小和响应时间有着很强的正比关系，比例系数是 0.94，如图 15-1 所示。这很有意思——向页面中添加更多的组件或更大的组件会使它更慢。对于任何需要进行性能改善的 Web 页面来说，在开发过程中持续对页面大小和响应时间进行测绘是一种很值得去做的分析。

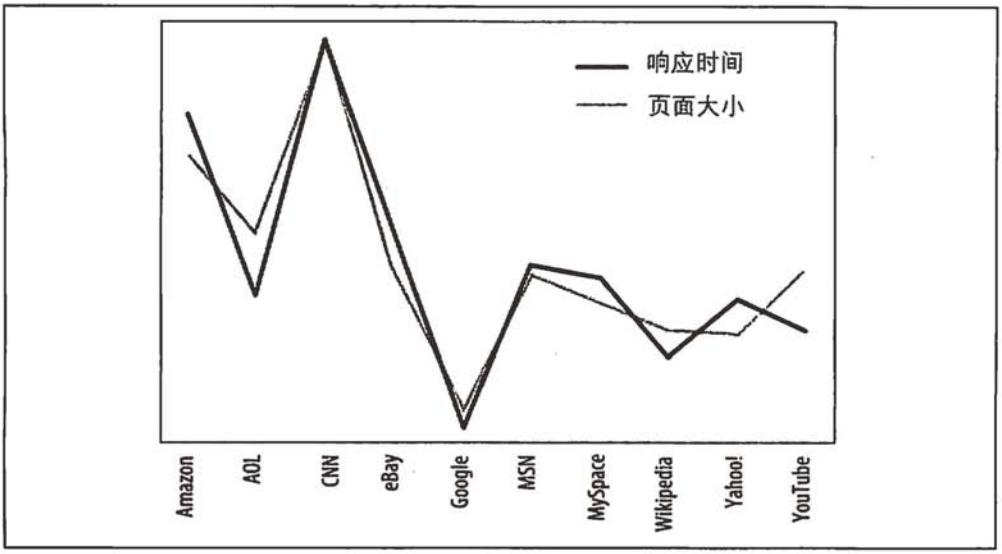


图 15-1：页面大小和响应时间的关系

YSlow 等级是页面响应时间的强指示器，如图 15-2 所示。高（好的）YSlow 等级表示页面构建良好，它既快又小。低（差的）YSlow 等级的页面可能会又慢又大。由于 YSlow 等级和响应时间及页面大小是成反比的，因此图 15-2 中绘制的是 YSlow 等级的倒数。YSlow 等级通常按照 A、B、C、D 或 F 给出，但在字母等级背后还会有一个 0~100 区间的数字得分。

Yahoo!不是很符合这个曲线，它具有第二好的 YSlow 等级（它的得分是 95，属于 A，比 Google 稍慢，Google 是 A，完美的 100 分）和响应时间，尽管它在页面大小中排名第四。Yahoo!的主页团队是这些性能最佳实践的长期消费者，因此能够得到很好的 YSlow 得分，并从页面中榨取更多的速度。Amazon 的 YSlow 等级也没有反映出页面大小和响应时间的一般关系规律。其中的主要原因是它们在页面中使用了大量的图片（大约 74 个图片）。YSlow 没有减去图片的点数，因此 Amazon 页面的得分很好，但性能较差。

一般来说，我们可以看到遵守这些最佳实践规则可以产生更快的页面反应速度。YSlow 等级的倒数和响应时间之间的比例系数是 0.76，也是很强的正比关系。这就是我在 Yahoo! 产品团队的工作经验。随着页面的改变，一次一次地应用这些规则，响应时间也一次一次

www.TopSage.com

大家网 TopSage.com

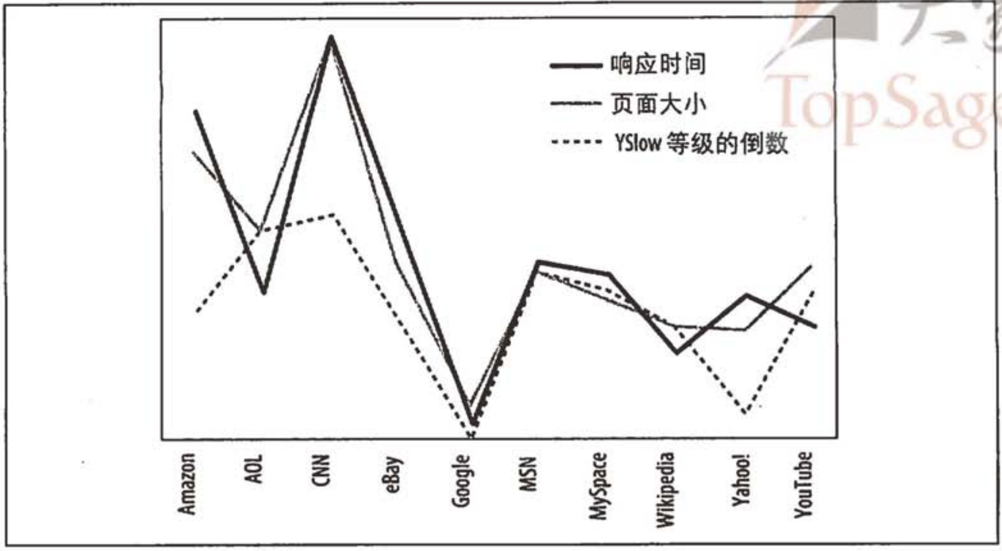


图 15-2：YSlow 等级与页面大小和响应时间的关系

地变快。在下一节“如何进行测试”中，我将介绍进行这些分析时使用的工具和测量手段。在这之后，我们进入对十大网站进行的性能分析。

如何进行测试

How the Tests Were Done

对十大网站进行剖析可以看出现实世界中的页面是如何遵守性能最佳实践规则的。进行这类分析的一个问题在于，待分析的目标是一个移动靶——这些网站都在持续地变化着。例如，在我进行分析的期间，一个网站从 IIS 切换到了 Apache。有可能——并且很可能——我所分析的页面并不是今天你访问这个网站时所看到的。理想情况下，你找到的页面将已经实施了这里强调的一些建议和其他一些最佳实践，并且性能更好、加载更快。

HTTP 请求图表使用 IBM Page Detailer（*http://alphaworks.ibm.com/tech/pagedetailer*）生成。这是我最喜欢的抓包工具。它能在所有 HTTP 客户端上使用。我喜欢 IBM Page Detailer 展现 HTTP 请求与对应 HTML 文档关联时的展现方式。使用该 HTTP 图表更容易指出组件下载时的瓶颈。图中的横条是带颜色的，指出了下载的组件的类型。

响应时间的测量使用的是 Gomez 的 Web 监视服务（*http://www.gomez.com*）。这里的响应时间被定义为从请求初始化完毕到页面的 `onload` 事件被触发所经过的时间。每个 URL 都在低带宽（56 KB ~512KB）下测试了数千次，这里展示的是平均值。

www.TopSage.com

我使用 Firebug（*http://www.getfirebug.com*）分析各个页面中的 JavaScript 和 CSS。对于任何一个前端工程师来说，Firebug 都是一个很重要的工具。它最强大的功能是能够调试 JavaScript 代码，不过这只是其中的一小部分功能。Firebug 还提供了检查 DOM、调整 CSS、执行 JavaScript 和浏览页面 HTML 等功能。

用于分析这些页面性能的主要工具是 YSlow（*http://developer.yahoo.com/yslow*）。我为 Yahoo! 的开发团队构建了 YSlow，帮助他们指出哪些改动能够带来最好的性能改进。Joe Hewitt ——Firebug 的作者——为将 YSlow 和 Firebug 进行集成提供了支持。这对于已经在开发过程中使用 Firebug 的前端工程师来说是一个理想的组合。

YSlow 通过遍历页面的 DOM 找到页面中所有的组件。它使用 `XMLHttpRequest` 找到每个组件的响应时间及 HTTP 响应头。这些信息，以及通过解析页面的 HTML 收集的其他信息一起用于针对每个规则进行评分，如图 15-3 所示。整体 YSlow 等级是每个规则得分的加权平均值。YSlow 还提供了其他工具，包括页面组件的汇总和使用 JSLint（*http://jslint.com*）对页面中所有 JavaScript 进行的分析。

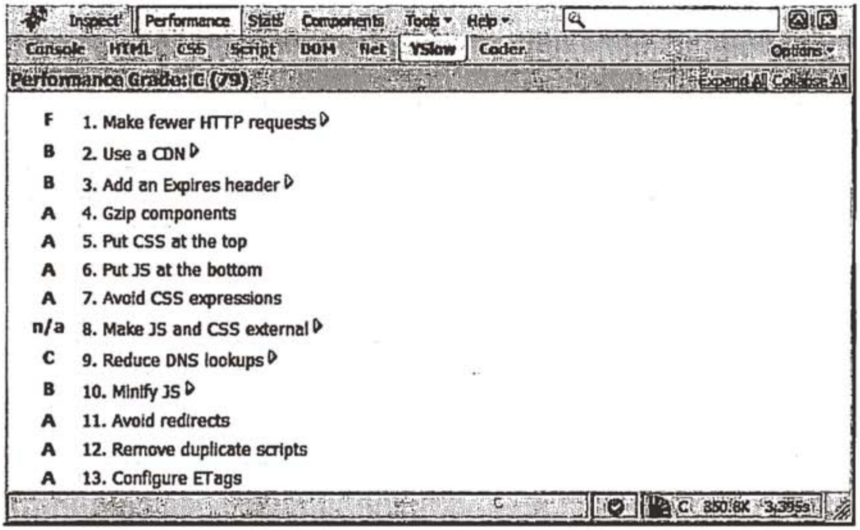


图 15-3：YSlow

www.TopSage.com

大家网
TopSage.com

Amazon

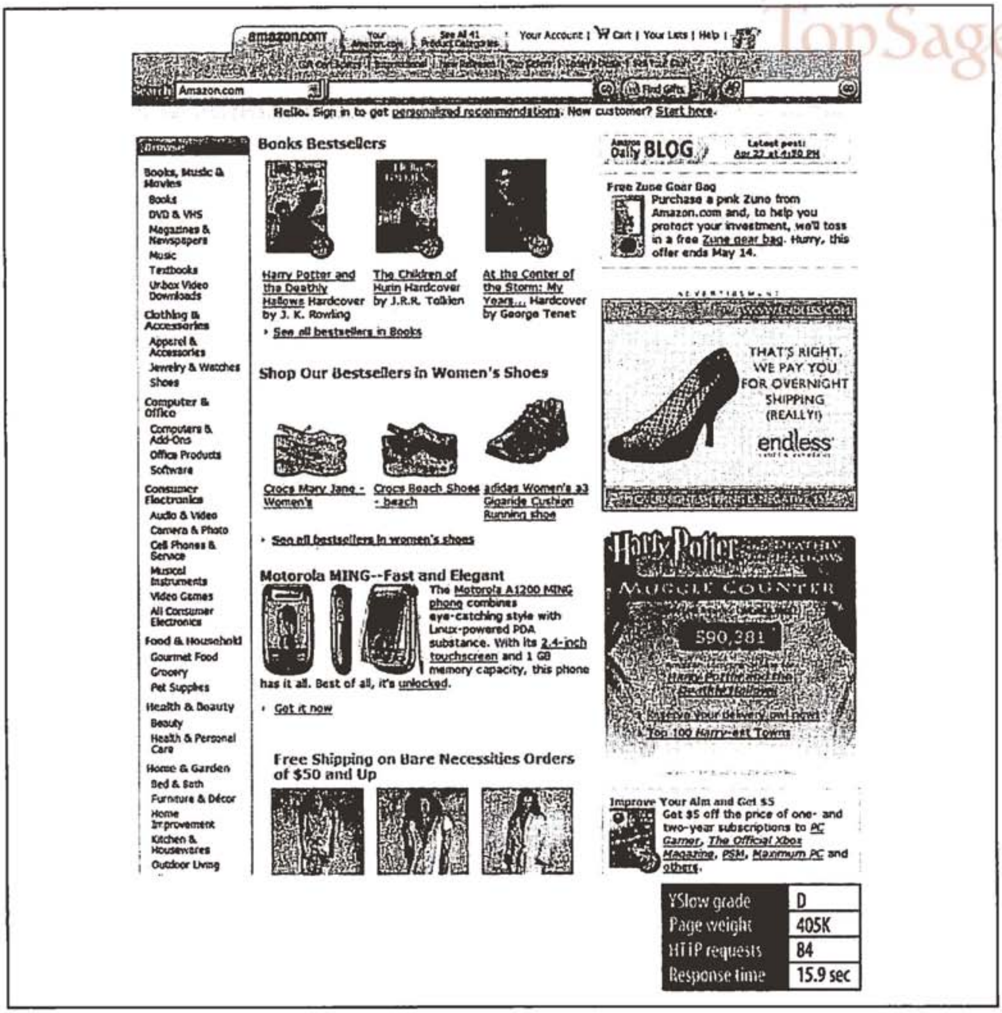


图 15-4：http://ww.amazon.com

Amazon（*http://www.amazon.com*）（如图 15-4 所示）是一个相对较大的页面，页面总大小为 405KB，有 84 个 HTTP 请求。由于页面中有这么大这么多的组件，为其组件添加长久的 `Expires` 头（规则 3）将可以带来最大的性能改进。84 个组件中只有 3 个有 `Expires` 头。他们只使用了 1 个样式表和 3 个脚本。这些脚本一个接一个地下载，因此一种简单的改进就是将它们合并为一个单独的 HTTP 请求。样式表和脚本应该经过压缩。3 个脚本都被最大程度地精简了，但如果移除所有的注释和无关的回车符还能进一步节省时间。

www.TopSage.com

即便 YSlow 指出了性能改进，但页面中如此大量的图片（74 个）绝对是个挑战。这些图片中的 19 个是在 CSS 规则中用作背景的。将这些图片转为 CSS SPRITES 可以将 HTTP 请求的总数从 88 个减少到 66 个。

TopSage.com

观察图 15-5 的瀑布图中的 HTTP 请求子集，我们可以看到由于请求的所有图片都来自于相同的主机名，因此只能并行下载两个图片，这增加了页面加载时间。将数量如此多的图片分别放到两个主机名下，可以使并行下载的数量加倍，显著地降低最终用户响应时间。选择数量 2 是第 6 章中“并行下载”一节建议的数量。

www.TopSage.com

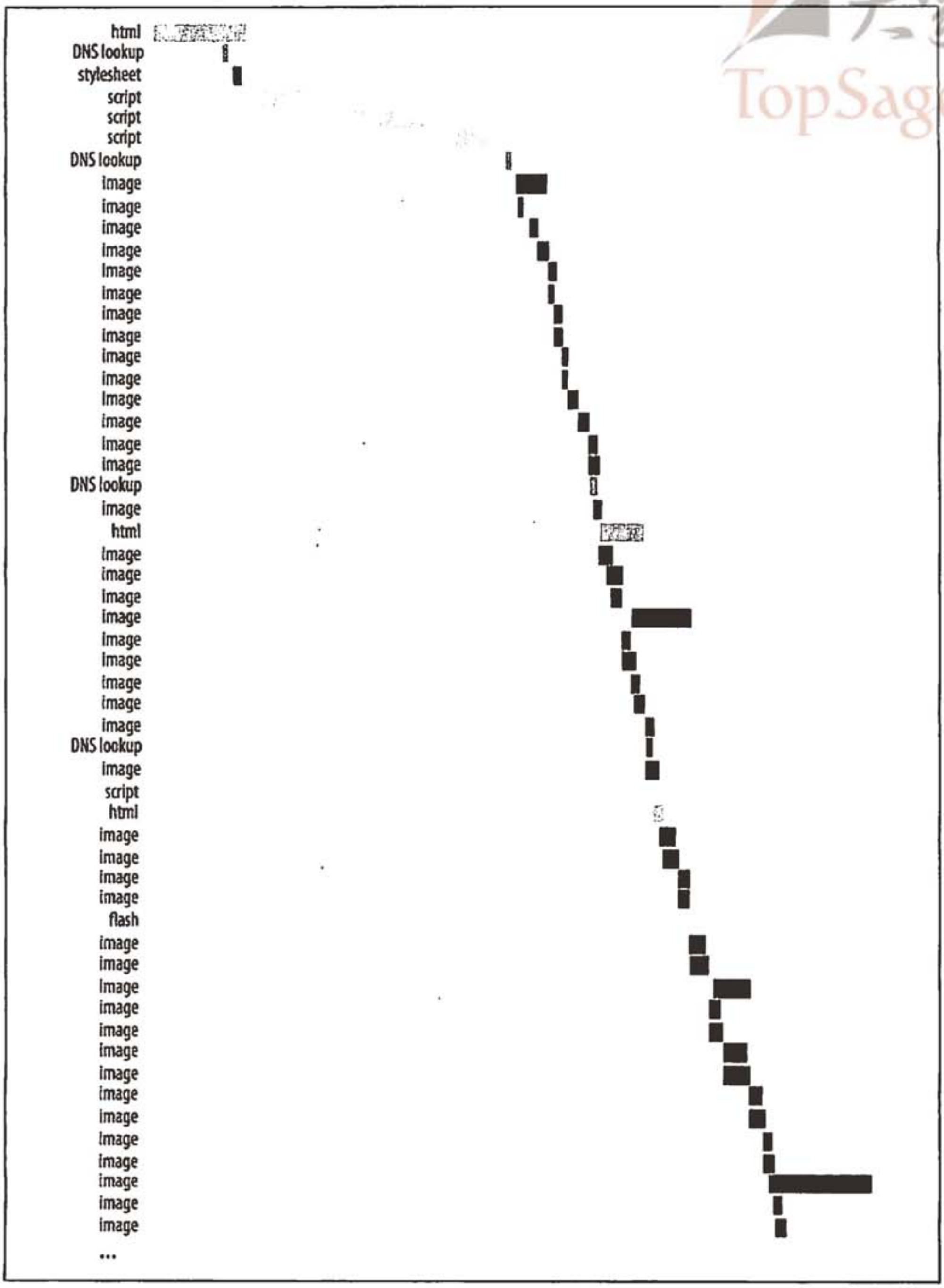


图 15-5：Amazon HTTP 请求

www.TopSage.com

大家网 TopSage.com

AOL

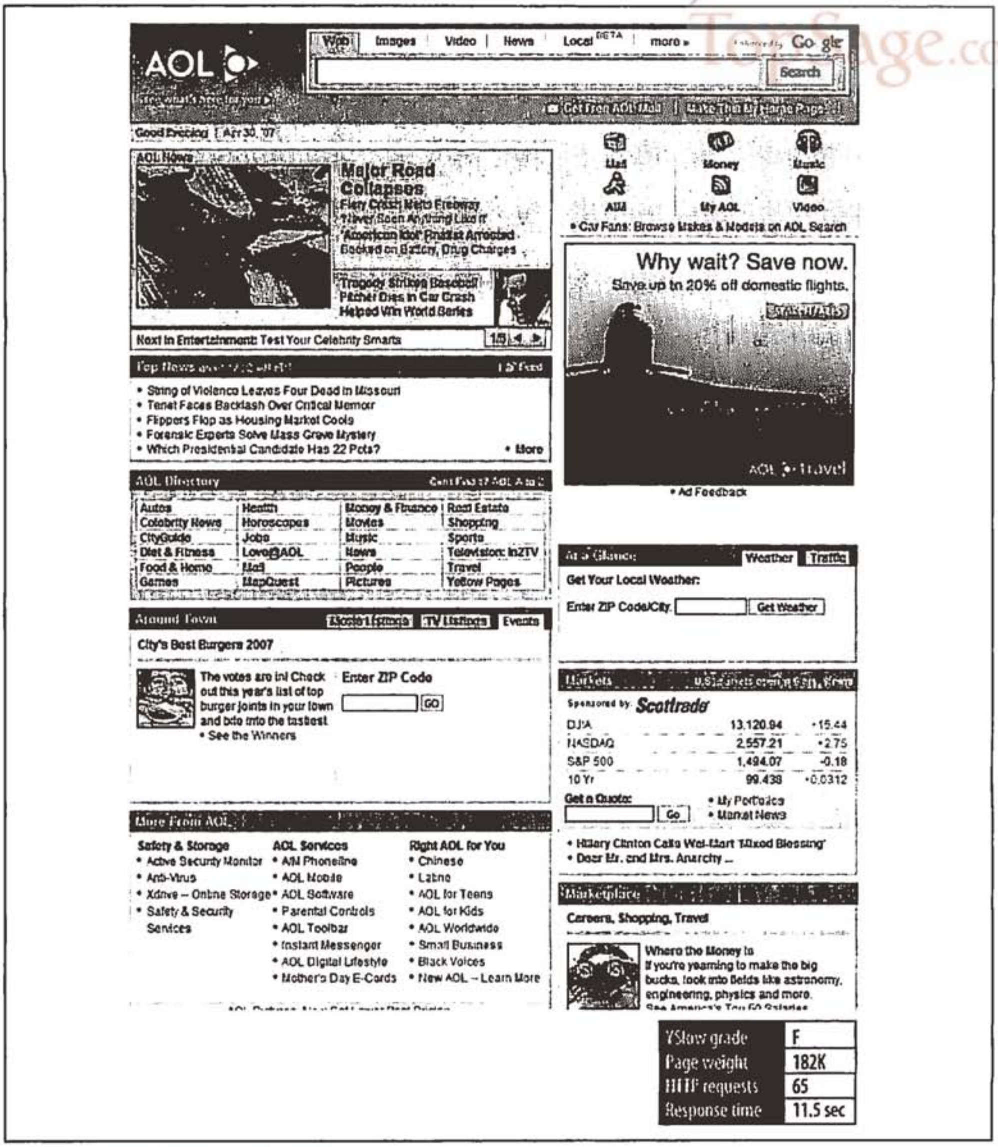


图 15-6：http://www.aol.com

AOL（*http://www.aol.com*）（如图 15-6 所示）的 HTTP 请求的前半部分最大程度地进行了并行下载，但在后半部分中，HTTP 请求是顺序进行的（如图 15-7 所示）。因此，页面加载时间也随之增加了。这里有两个有趣的实现细节——降级到 HTTP 1.0 和多重脚本。

www.TopSage.com

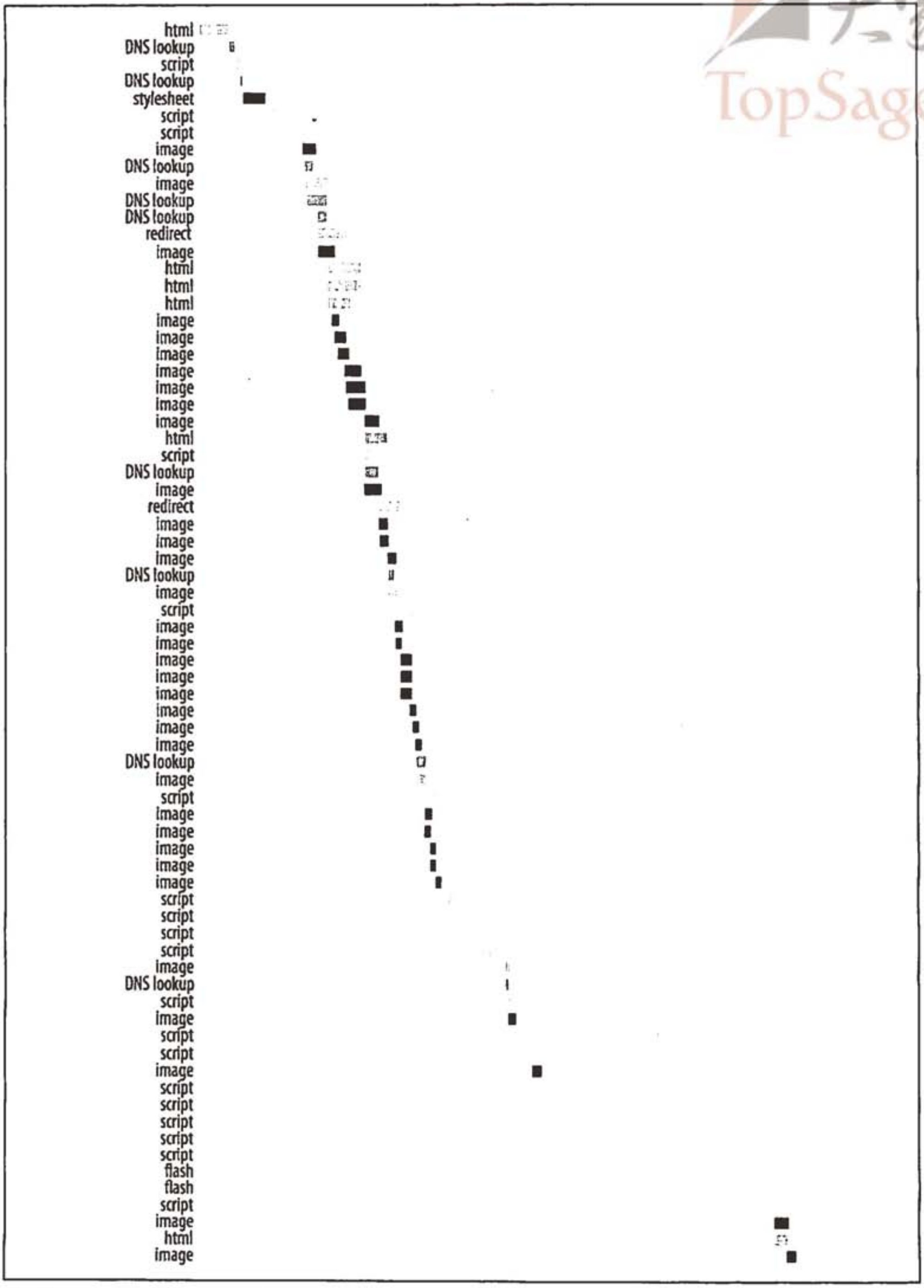


图 15-7：AOL 的 HTTP 请求

www.TopSage.com

在前半部分，也就是并行化较好的部分，响应从 HTTP 1.1 降级为了 HTTP 1.0。我通过观察 HTTP 头在请求时使用的规范是 HTTP 1.1 而响应状态是 HTTP 1.0 发现了这一点。

TopSage.com

```
GET /_media/aolp_v21/bctrl.gif HTTP 1.1
Host: www.aolcdn.com
```

```
HTTP 1.0 200 OK
```

对于 HTTP 1.0，规范建议每个主机名并行下载四个组件，而 HTTP 1.1 的建议是每个主机名并行下载两个组件。Web 服务器通过在响应中对 HTTP 版本进行降级，得到了更高的并行度。

通常，我只在过时的服务器配置中看到过这种结果，但这也可能是有意而为之的，为的是增加并行下载的数量。在 Yahoo!网，我们对此进行了测试，发现 HTTP 1.1 的整体性能更好一些，因为 HTTP 1.1 在默认情况下提供了持久化连接（参见绪言 B 中的“Keep-Alive”一节）。

AOL 的 HTTP 流量的后半部分大多没有进行并行下载，因为其中很多请求是脚本。正如第 6 章介绍的那样，当浏览器下载外部脚本时，所有其他下载都将被阻塞。结果就是数量本来不多的脚本占用了比并行下载更长的时间段。

这些脚本看起来是用于广告的，但脚本的插入看来很低效。这些脚本成对出现，第一个脚本包含——

```
document.write('<script type="text/javascript" src="http://twx.doubleclick.net/adj/
TW.AOLCom/Site_WS[snip...]script>\n');
```

这导致了从 *http://twx.doubleclick.net* 下载第二个脚本。其中包含了广告的内容——

```
document.write('<!-- Template Id = 4140 Template Name = AOL - Text - WS Portal ATF DR
2-line (291x30) -->\n<B>Free Credit Score</B>[snip...]');
```

有 6 个广告使用了这种方式，总共有 12 个外部脚本必须下载。如果每个广告只需调用和下载一个脚本，可以省掉 6 个 HTTP 请求。这些额外的请求对页面加载时间有着显著的影响，因为这些脚本阻塞了其他下载。

可以产生巨大改进的其他方面还有——

规则 3——添加 Expires 头

30 多张图片没有被缓存，因为它们没有 `Expires` 头。

www.TopSage.com

大家网 TopSage.com

规则 4——压缩组件

一个样式表和 20 个脚本没有经过压缩。

规则 9——减少 DNS 查找

使用了 11 个域名，这意味着很可能由于额外的 DNS 查找而导致延迟。

页面中还使用了 4 个信标，其中 3 个是在页面加载完毕后发送的。这些信标提高性能的原因是它们都使用了“204 No Content”状态码。该状态码对信标来说非常理想，因为信标不包含完整的正文，这样做可以使响应更小。

www.TopSage.com

大家网 TopSage.com

CNN

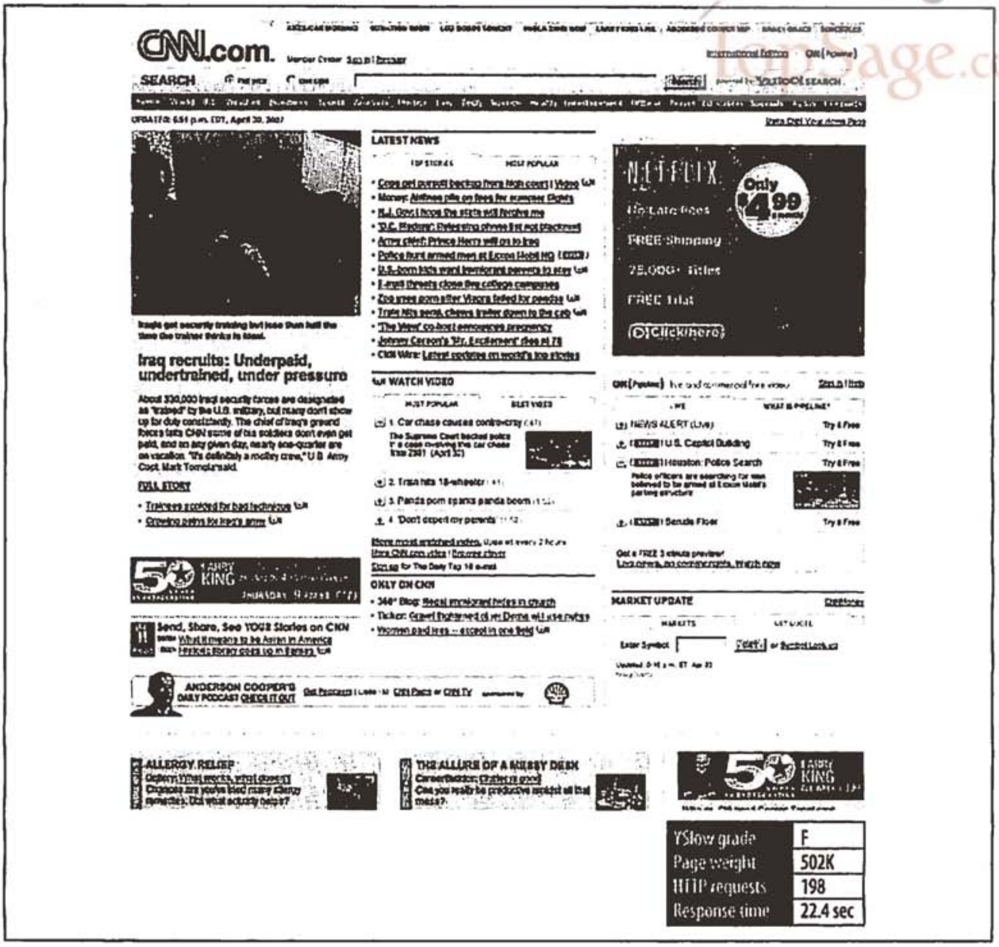


图 15-8：http://www.cnn.com

CNN（*http://www.cnn.com*）（如图 15-8 所示）在页面总大小（502KB）和 HTTP 请求数量（198!）上都是十大网站中最大的，其主要原因是它使用图片来显示文字。例如，图片 *http://i.a.cnn.net/cnn/element/img/1.5/main/tabs/topstories.gif* 就是文字 “Top Stories”，如图 15-9 所示。

图 15-9：将文字呈现为图片

www.TopSage.com

只包含文字的图片有 70 多个。用图片显示文字可以得到用文本字体无法实现的自定义外观。作为代价——可以从下载统计中看到——页面的大小和 HTTP 请求的数量增加了，导致用户体验变得缓慢。同时，国际化也变得更加有挑战，每一种翻译都需要一组新的图片。规则 1 告诉我们，在迈向更好的性能时，减少组件的数量是最重要的一步。用文本替换图片会给该页面带来性能的最大改进。

类似地，用作 CSS 背景的图片有 16 个。如果能将他们组合到少量的 CSS Sprites 中，如第 1 章所说，可以省掉 10 个或更多个 HTTP 请求。将 10 个分立的 JavaScript 文件合并到一起能省掉另外 9 个 HTTP 请求。

更进一步，页面中超过 140 个的组件没有 `Expires` 头，因此无法被浏览器缓存（规则 3）。没有任何样式表或脚本是经过压缩的（规则 4），而且很多脚本没有经过精简（规则 10）。样式表总计 87KB，脚本是 114KB，因此压缩和精简能够显著减小页面大小。超过 180 个组件具有来自于 Apache 的默认 ETag。正如第 13 章介绍的那样，这意味着当进行条件 GET 请求时，这不会产生更有效的 304 状态码。此处这种情况尤其严重，因为很多组件没有长久的 `Expires` 头，因此必须进行验证。

www.TopSage.com

大家网 TopSage.com

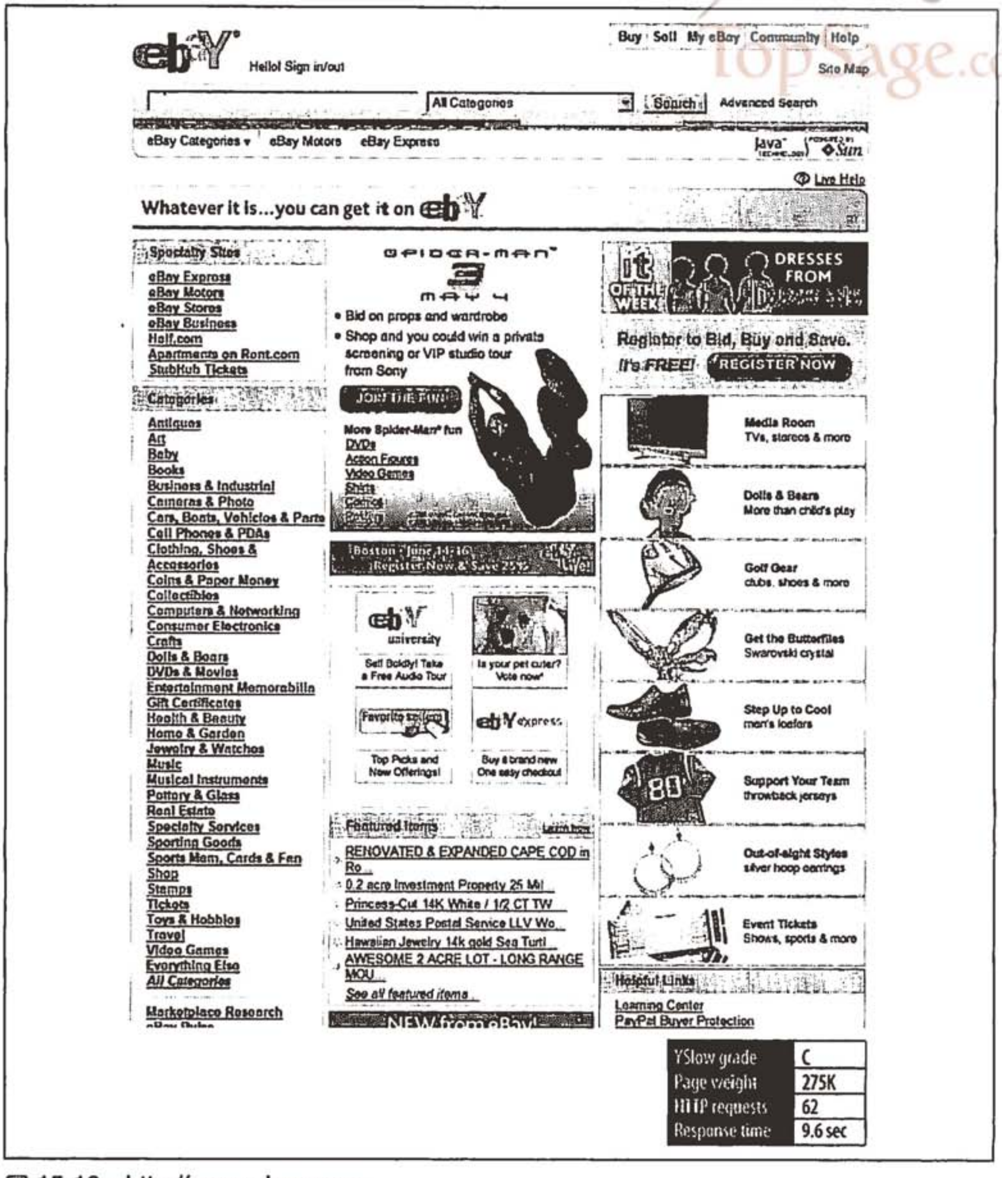


图 15-10：http://www.ebay.com

eBay（*http://www.ebay.com*）（如图 15-10 所示）的 YSlow 等级非常接近 B。只须再做一点工作它的性能就会更好。其主要问题与规则 1、3、9 和 13 相关。

www.TopSage.com

规则 1——产生更少的 HTTP 请求

eBay 页面有 10 个脚本和 3 个样式表。一个简单的修正方法就是使用第 1 章介绍的合并技术。有 4 个脚本是在接近文档顶部加载的，3 个位于页面底部。这些可以合并为一个顶部脚本和一个底部脚本，将脚本从 10 个减少到 5 个。3 个样式表也是紧挨着加载的，也应该进行合并。

规则 3——添加 Expires 头

eBay 页面只有一个脚本和一个样式表具有 `Expires` 头，但是只有 9 个小时。根据 `Last-Modified` 日期来看，样式表 3 天不会改变，脚本是 24 天。这样做有助于在过期时改变内容，但对于该网站的用户数量，使用长久的 `Expires` 头让这些文件缓存可能会更好一些。另外，还有 5 个 IFrame 没有使用 `Expires` 头。这些 IFrame 用于插入广告图片，而这些图片有的也没有 `Expires` 头。

规则 9——减少 DNS 查找

eBay 页面使用了 9 个不同的域名。通常，域名数量多是因为包含了很多第三方广告的域名，但是在这里，有 7 个域名是与 eBay 相关的，只有 2 个用于第三方广告。

规则 13——配置 ETag

来自 IIS 提供的 52 个组件使用了默认的 ETag。第 13 章曾介绍过，这会导致组件比必要情况下下载得更加频繁。由于这些组件的过期时间最多为未来 45 天，因此这种情况会更加恶化。当组件过期并进行条件 GET 请求时，ETag 会破坏得到更快的“304 Not Modified”响应的机会，最后导致即便组件已经存在于用户的磁盘上，还要将整个组件传送到客户端。

使用 IFrame 提供广告的方式值得探讨。IFrame 实现了广告与实际页面的彻底分离，使得两者的团队和系统能够互不依赖。缺点是每个 IFrame 都带来一个额外的 HTTP 请求，通常（例如这里）它是不可缓存的。

使用 IFrame 来提供广告是合理的，因为广告通常包含自己的 JavaScript 代码。如果广告内容来自于第三方并且包含 JavaScript，将其放到 IFrame 中也就把 JavaScript 放到了沙箱中，可以带来极高的安全性（第三方的 JavaScript 代码无法访问 Web 页面的命名空间）。然而，在 eBay 的页面中，IFrame 中提供的广告并不包含 JavaScript。此外，只有一个包含第三方内容。在生成 HTML 页面的时候插入这些广告可以省掉 5 个 IFrame HTTP 请求。

另一个额外的改进是可以将大量的图片分别放到两个主机名下。41 个图片中有 36 个来自于 *http://pics.ebaystatic.com*。在 HTTP 1.1 中，每个主机名只有两个组件能够并行下载（参见第 6 章）。这对 HTTP 请求的并行度有着负面影响（参见图 15-11）。

www.TopSage.com

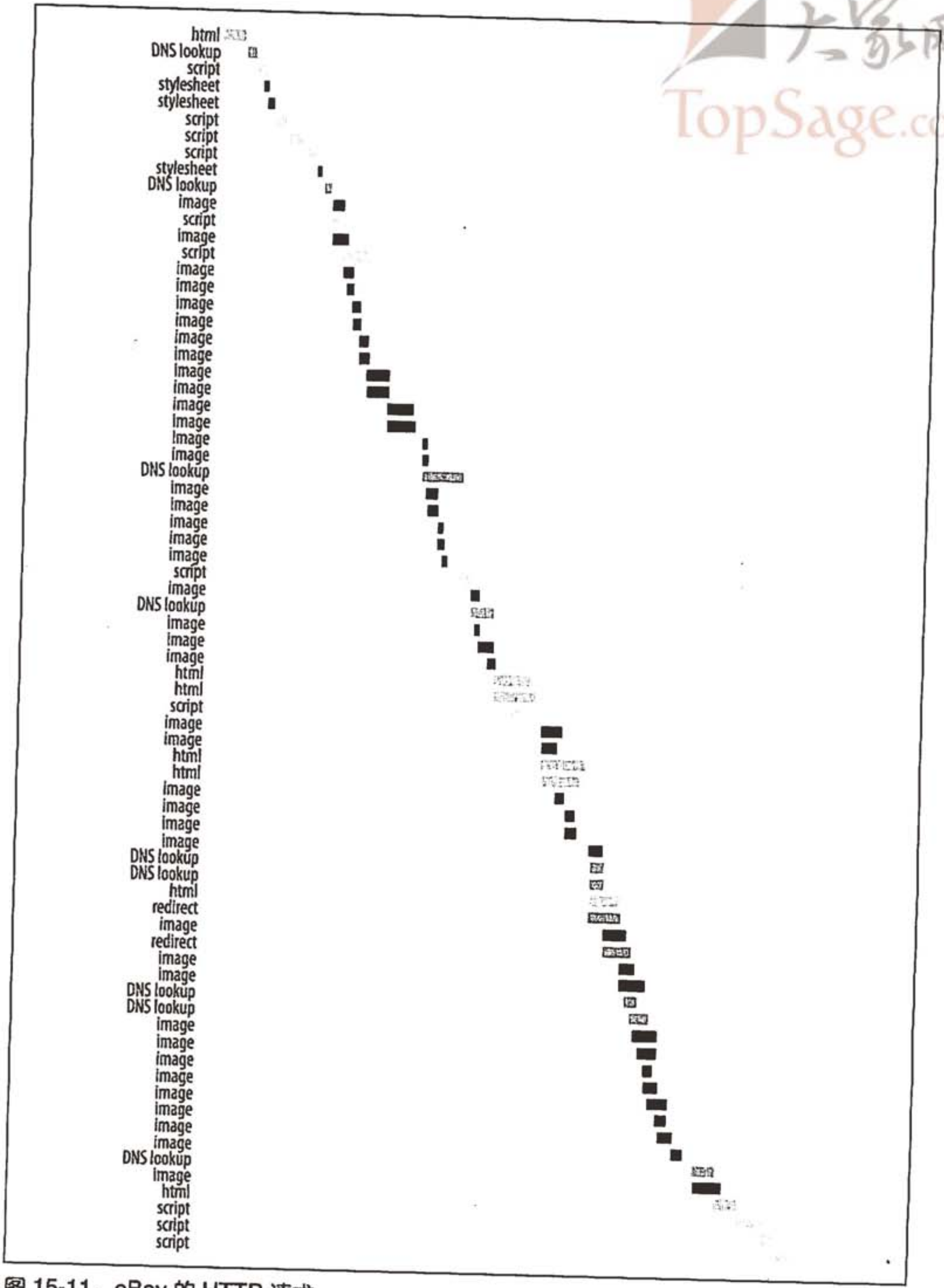


图 15-11：eBay 的 HTTP 请求

www.TopSage.com

这 36 个图片中的很多都是在图示的中部下载的，你可以清晰地看到一个阶梯的形状，每次只有两个请求。如果将它们放到两个主机名下，例如 *http://pics1.ebaystatic.com* 和 *http://pics2.ebaystatic.com*，则可以并行下载 4 个图片，从而缩减了整体页面加载时间。这需要在将图片分别放到多个主机名下和减少 DNS 查找（规则 9）之间进行性能权衡，但在这里——下载 36 个图片，每次 4 个——值得进行一次额外的 DNS 查找。

其良好的性能特征在于，有 3 个脚本是在页面底部进行下载的。这些脚本与用户在 eBay 的“收藏夹”相关，而且对于呈现页面来说可能并不是必需的。eBay 遵守了在底部加载脚本的推荐实践，第 6 章介绍了这是有价值的，因为它不会阻塞下载和呈现。

www.TopSage.com

大家网 TopSage.com

Google

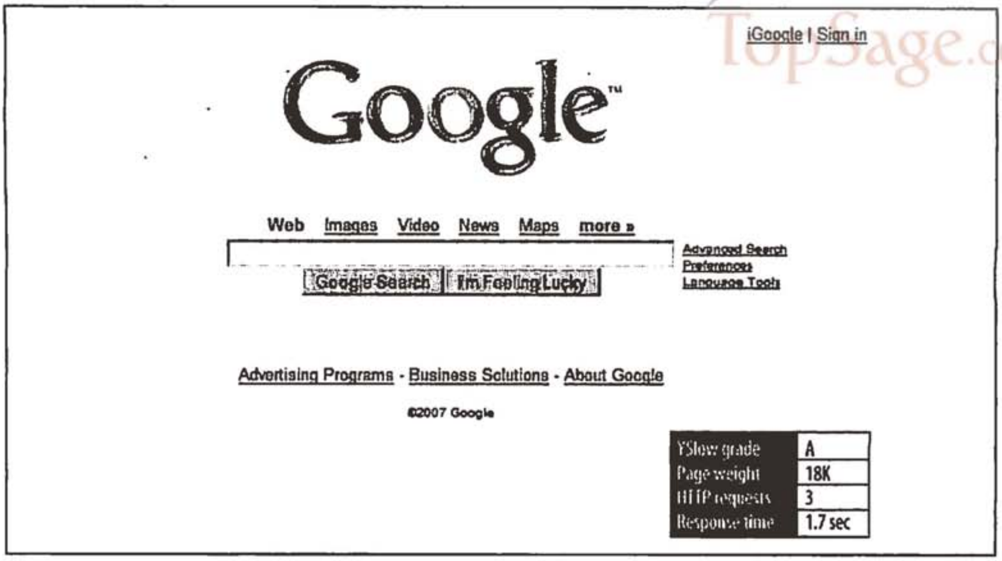


图 15-12：http://www.google.com

Google（页面如图 15-12 所示）以其简单和快速的页面设计著称。其首页——*http://www.google.com*——的页面总大小只有 18KB，并且只发送 3 个 HTTP 请求（HTML 文档和两个图片）。然而，即便是这样简单的页面，也有很多性能优化工作值得去做。

Google 页面只有 3 个 HTTP 请求，但图 15-13 显示了 5 个 HTTP 请求。

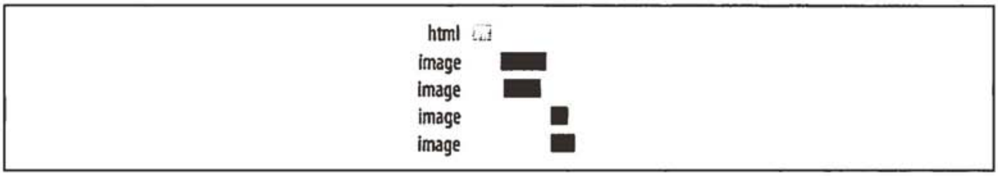


图 15-13：Google 的 HTTP 请求

额外的两个 HTTP 请求并不是页面的一部分。一个是 *http://www.google.com/favicon.ico*（参见图 15-14）。Favicon 是用来与 URL 关联的可视化图片。它们显示在浏览器顶部的 URL 附近、书签或收藏夹列表的 URL 附近和标签（对于支持标签的浏览器来说）上。浏览器会在第一次加载一个网站时获取它们。如果网站没有 Favicon，就会使用默认图片。

www.TopSage.com

TopSage.com

图 15-14：http://www.google.com/favicon.ico

第二个额外的请求用于 *http://www.google.com/images/nav_logo3.png*，如图 15-15 所示。这是一个 CSS SPRITES，与第 1 章所介绍的一组图片组合。我之所以说它不是页面的一部分，是因为它是在 Google 首页加载完毕后，作为 `onload` 事件的一部分进行的——

```
onload="sf();if(document.images){new Image().src='/images/nav_logo3.png'}"
```

对 `sf()` 函数的调用将输入焦点放到了搜索框中。第二条语句使用 `new Image()` 创建了一个图片对象。该图片对象的 `src` 属性被设置为 `/images/nav_logo3.png`。这是一种典型的动态加载图片的方式，但除了一件事——新的图片没有赋给它任何变量。这样稍后在该页面中很难访问到这个图片。但这没有问题，因为该页面并没有使用该图片的意图。下载 *nav_logo3.png* 是因为可以预料到用户将要访问的页面。注意这个 CSS Sprites 是如何包含在搜索“下一页”和“上一页”图片中的。它还包含了用于其他页面的图片，如结账和购物车。

图 15-15：http://www.google.com/images/nav_logo3.png

这被称作预加载（*Preloading*）。当能够断定用户将要访问的下一个页面时，可以在后台下载后续页所需的组件。但是在 Google 的页面中，存在一个问题——任何后续页面都不会使用 *nav_logo3.png*。在从 *http://www.google.com* 提交一个搜索后，用户会去往 *http://www.google.com/search*。该搜索结果页加载了 *http://www.google.com/images/nav_logo.png*（“logo”后面没有“3”）。如图 15-16 所示，*nav_logo.png* 和 *nav_logo3.png* 很像。它也是一个 CSS Sprite。

图 15-16：http://www.google.com/images/nav_logo.png

www.TopSage.com

既然搜索结果页不使用 *nav_logo3.png*，为什么 Google 主页还要预加载它呢？它很可能是为其他 Google 站点预加载的，但我访问了 *http://froogle.google.com*、*http://catalogs.google.com*、*http://books.google.com* 和很多其他网站，没有一个使用了 *nav_logo3.png*。可能这是之前的设计留下来的，还没有被清除掉。它可能也预示着将来的网站集成策略（所以使用"3"）。尽管这明显浪费了 Google 主页的下载量，却也无伤大雅。预加载是提高网站后续页面加载速度的一种很好的策略。

Google 主页的另一个有趣的性能改进是对 SCRIPT DEFER 属性的使用。在第 6 章中，我介绍了当脚本阻塞了下载和呈现时，DEFER 属性并不能完全解决负面的性能影响。然而，那是关于外部脚本的，在这里，脚本是内联的——

```
<script type="text/javascript" defer><!--
function qs(el){...
function togDisp(e){...
function stopB(e){...
document.onclick=function(event){...
//-->
</script>
```

使用 DEFER 属性告诉浏览器继续呈现，稍后再执行 JavaScript，避免了可能的呈现延迟，但我从来没见过有谁将其用于内联脚本。在内联脚本上使用它的理由可能是 JavaScript 代码的解析和执行会延迟页面的呈现。但是这里有一个问题，在 SCRIPT 块之后，有一个超链接依赖 togDisp 函数来为"more"链接显示一个弹出式 DIV——

```
<a href="/intl/en/options/" onclick="this.blur();return togDisp(event)">more</a>
```

如果 DEFER 属性允许不执行 togDisp 函数的定义就呈现页面，就会产生一个竞态环境。如果"more"链接已经呈现出来，而用户在 JavaScript 执行之前单击了这个链接，就会发生错误。在内联脚本上使用 DEFER 是一个有待进一步研究的领域。

然而，这些建议已经超出了大多数网站所需的典型性能改进方法。Google 页面在 YSlow 中得到了完美的 100 分——它是 Internet 上最快的页面之一。

www.TopSage.com

MSN

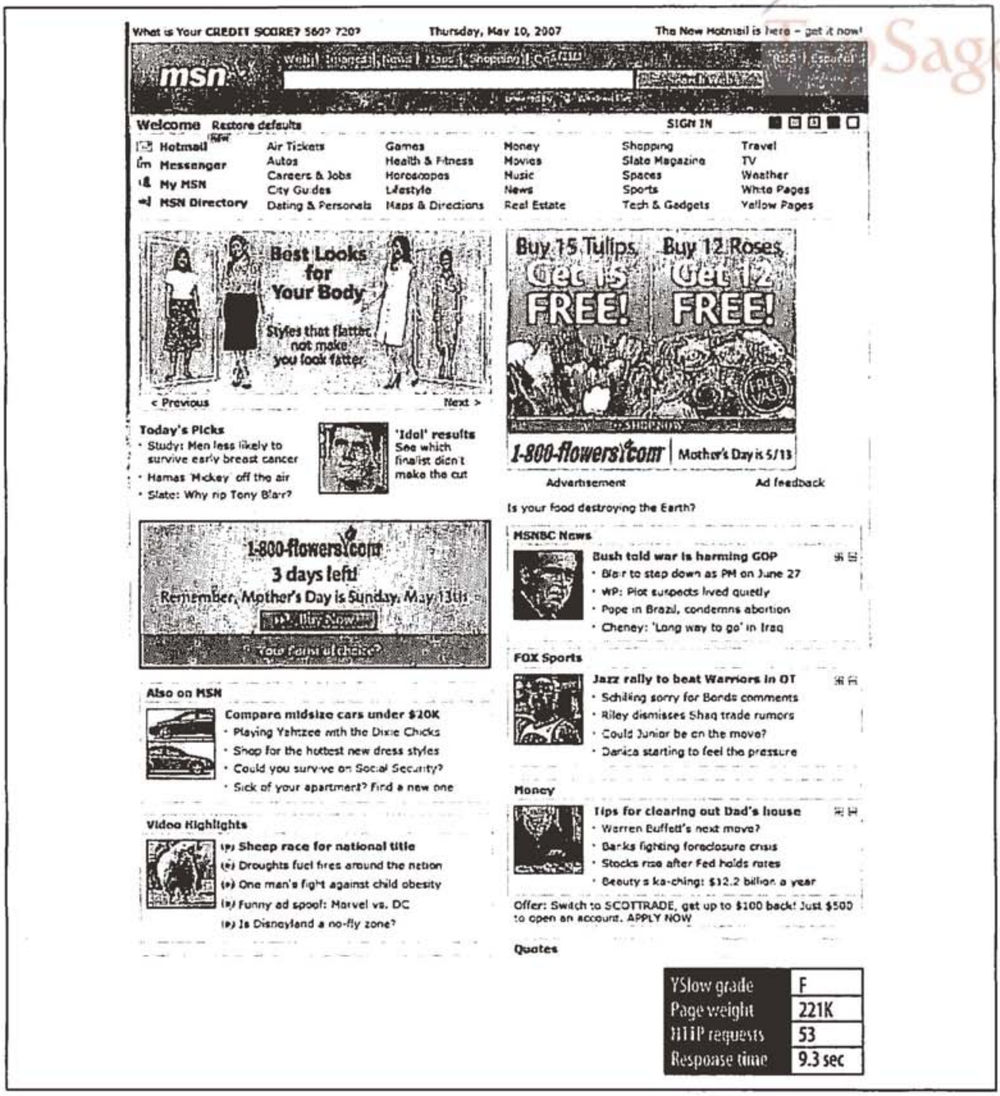


图 15-17：http://www.msn.com

MSN 主页（*http://www.msn.com*）（如图 15-17 所示）就其总大小和 HTTP 请求数量而言，居本章所检测的所有网站的中间地位。它没能遵守一些基本的性能指导方针，尤其是在插入广告的时候。然而，它具有很多正面的性能特征，这里分析的所有其他网站都不具备这些特征。我们从 MSN 如何插入广告开始讲起，因为在后面的很多建议中会提到它。

www.TopSage.com

MSN 使用 IFrame 向页面中插入了 5 个广告。前面曾和 eBay 一起讨论过，使用 IFrame 是消除广告系统和 HTML 页面生成系统之间依赖性的一种简单的方式。然而，每个 IFrame 都会导致一个额外的 HTTP 请求。在 MSN 中，每个 IFrame 的 `SRC` 属性都被设置为 `about:blank`，这不会生成任何 HTTP 流量。然而，每个 IFrame 都包含一个外部脚本，使用 JavaScript 和 `document.write` 语句向页面中插入广告。将广告系统和 HTML 页面生成系统集成在一起，这样不仅消除了这 5 个 HTTP 请求，而且只需要请求一个包含多个 `document.write` 语句的脚本。该 JavaScript 可以内联到 HTML 文档中。

规则 1——减少 HTTP 请求

MSN 主页有 4 个脚本（除了用于广告的脚本），其中 3 个加载时离得很近，应该进行合并。它还有超过 10 张 CSS 背景图片，这些可以使用 CSS Sprite 进行合并。

规则 3——添加 Expires 头

有一个脚本不能被缓存，因为它的过期日期被设置为过去。用于插入广告的 5 个脚本也具有过去的过期时间，因此它们也不能被缓存。这些 JavaScript 可能的确不能缓存，但如果有 HTML 页面自己来插入这些广告，就无需请求这 5 个外部脚本文件了。

规则 4——压缩组件

两个脚本和两个样式表没有进行压缩。另外，5 个用于提供广告的脚本也没有压缩。

规则 9——减少 DNS 查找

MSN 主页使用了 12 个域名。这比很多 Web 页面使用的都多，但稍后我们会讨论这在增加并行下载方面的优势。

规则 10——精简 JavaScript

用于提供广告的 5 个脚本没有经过精简。

规则 13——ETag——配置 ETag

页面中的很多组件拥有 ETag 并使用了 IIS 的默认格式。从不同服务器下载的相同的图片具有不同的 ETag，这意味着它们下载得比所需的要频繁。

MSN 主页还存在很多显著的性能优化——

- 它使用了一个 CSS Sprite（*http://stc.msn.com/br/hp/en-us/css/19/decoration/buttongs.gif*），十大网站中只有很少几家做到了这一点（另外两家是 AOL 和 Yahoo!）。该 CSS Sprite 如图 15-18 所示。

www.TopSage.com

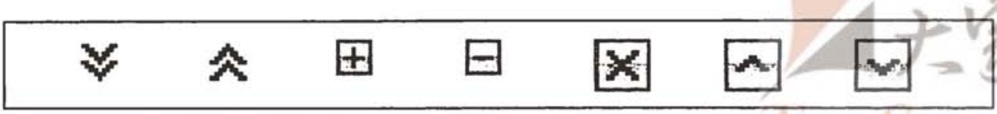


图 15-18：MSN 网站的 CSS Sprites 中存放的图片

TopSage.com

- 整个 HTML 文档都经过了精简。没有任何其他网站做到了这一点。
- 组件被分别放到多个主机名下，增加了并行下载，如图 15-19 所示。这是通过一种经过非常认真地考虑过的方式实现的——所有的 CSS 图片和页面中显示的所有其他图片都来自不同的主机名。

很明显，MSN 的员工中已经有人在关注性能的某些方面了。然而，如果将广告和 HTML 页面集成在一起，并修正一些 Web 服务器配置设置，还能为其页面带来更巨大的性能改善。

www.TopSage.com

大家网 TopSage.com

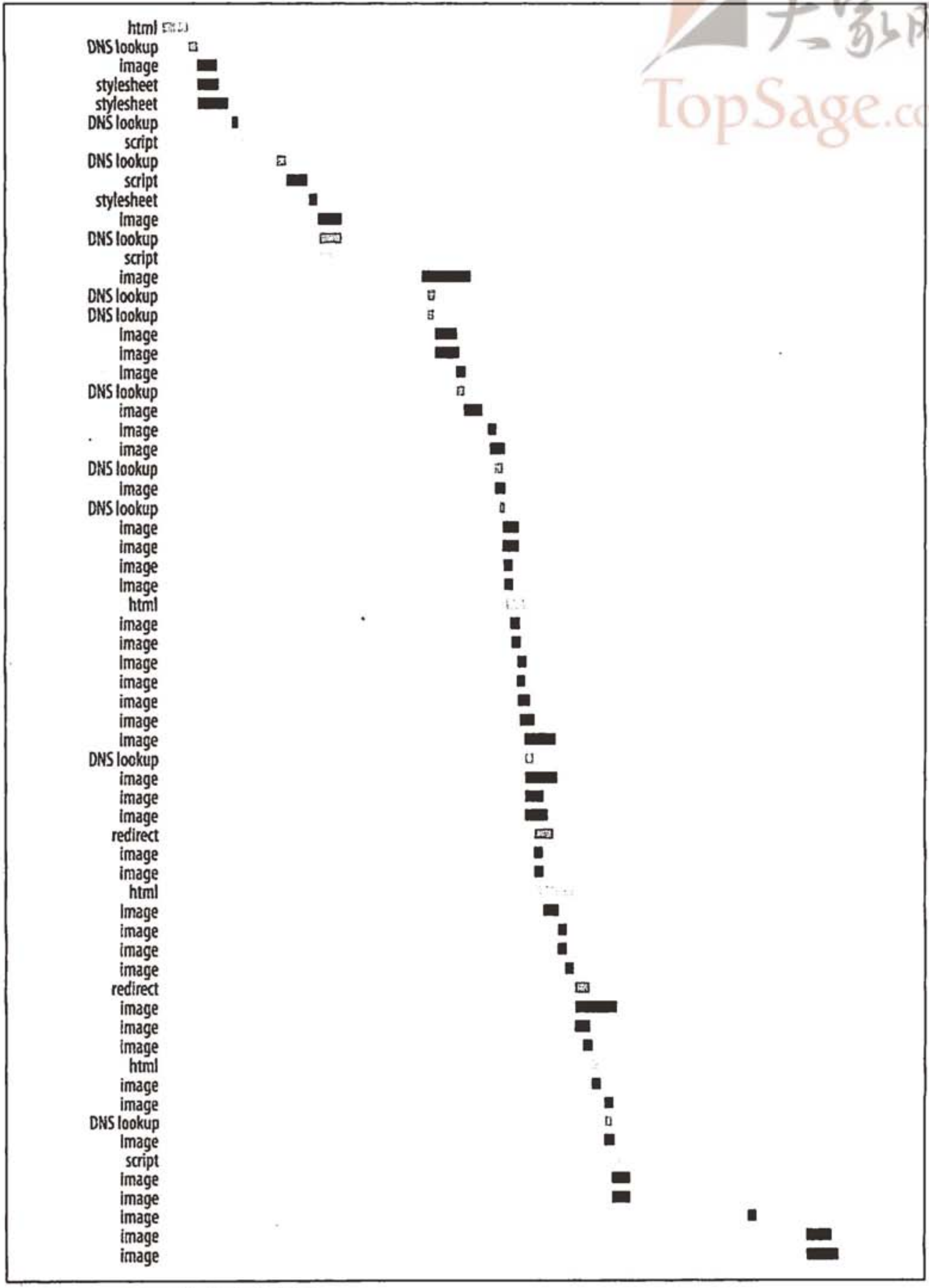


图 15-19：MSN 的 HTTP 请求

www.TopSage.com

MySpace

大家网 TopSage.com

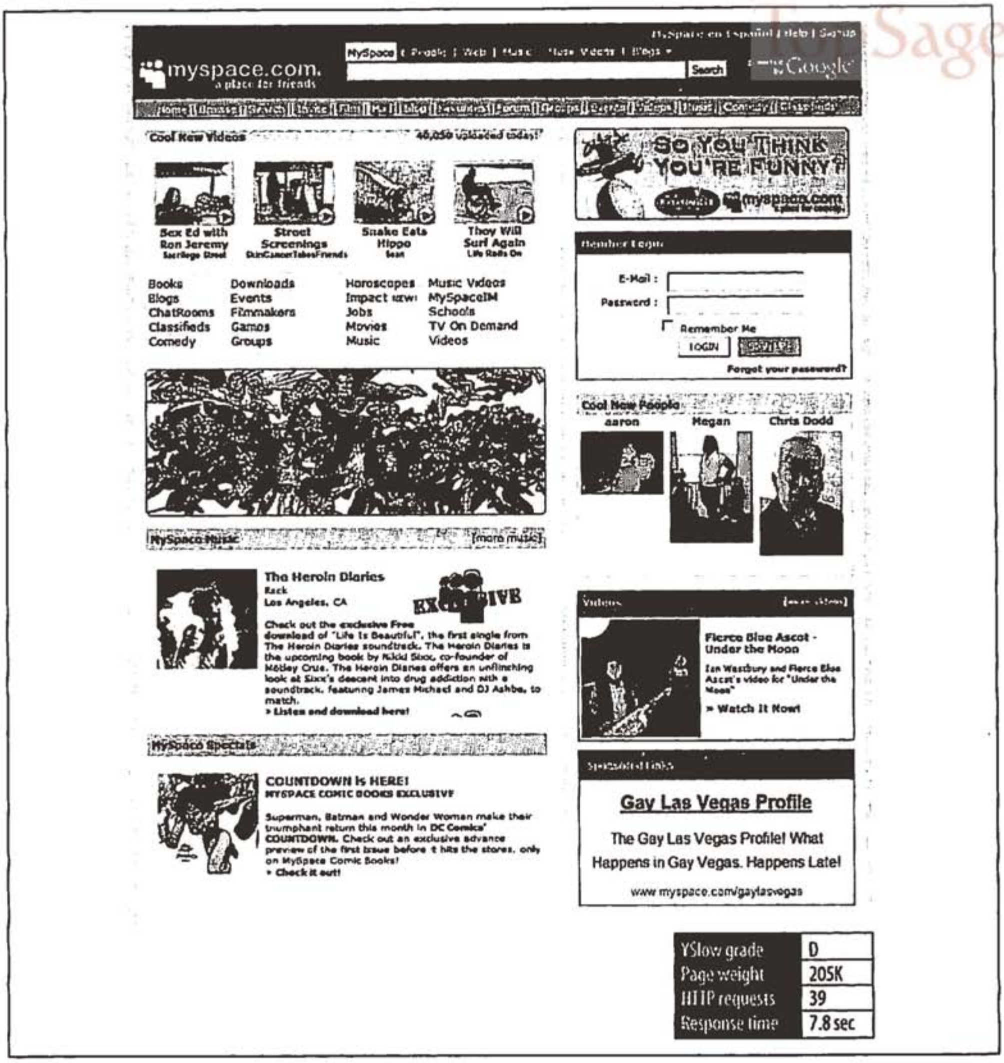


图 15-20：http://www.myspace.com

让包含由用户生成内容的网站达到最高的性能是一种挑战——其内容会很频繁地变化。然而，还是有一些简单的改变能够缩减 MySpace（*http://www.myspace.com*）（如图 15-20 所示）的响应时间的。

www.TopSage.com

大家网 TopSage.com

规则 1——减少 HTTP 请求

合并脚本和样式表可以减少 HTTP 请求的数量。该页面有 6 个脚本，其中 3 个都在页面顶端很近的位置下载，合并起来很容易。3 个样式表也在页面顶端附近加载，也是很容易合并的。

规则 3——添加 Expires 头

MySpace 页面中有超过一打的图片没有 `Expires` 头。页面中的某些图片无法利用 `Expires` 头的原因是可以理解的，因为它们变化得很频繁，如页面中的 new video 和 new people 部分。然而，有一些在每个页面都会使用的图片也没有 `Expires` 头。

规则 9——减少 DNS 查找

对于页面中使用的 10 个唯一域名来说，如果能省掉一些，一定能降低 DNS 查找产生的负面影响。

规则 10——精简 JavaScript

4 个脚本——总计超过 20KB——没有进行精简。

如图 15-21 所示，在页面中部 HTTP 请求的并行度很高，但在开始时存在着由脚本和样式表的阻塞行为带来的负面影响（第 6 章介绍了这种阻塞行为）。合并这些文件可以减少这种影响。因为是在 Firefox 中测量的这些 HTTP 请求，所以情况更为严重一些。除了脚本会阻塞并行下载（同时出现在 Firefox 和 Internet Explorer 中）外，样式表也会阻塞并行下载（仅出现在 Firefox 中）。所以，合并脚本和样式表对于 Firefox 和 Internet Explorer 来说都能改善性能。

www.TopSage.com

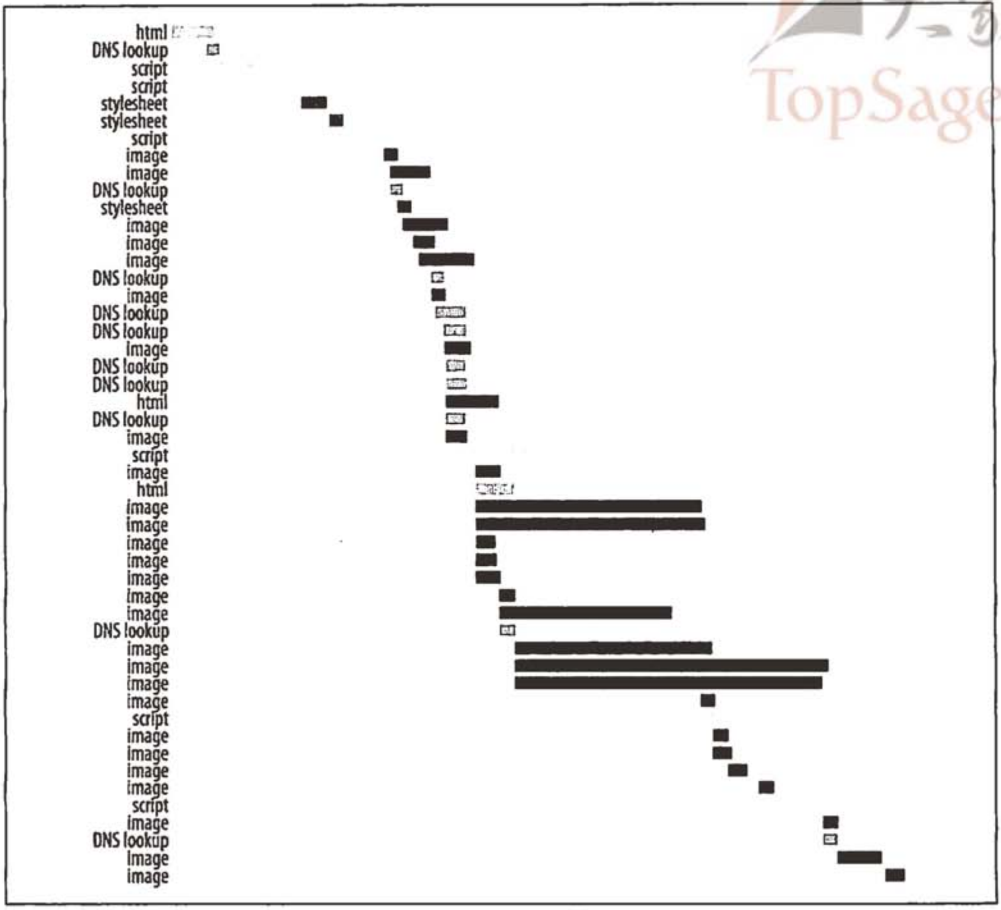


图 15-21：MySpace 的 HTTP 请求

www.TopSage.com

大家网 TopSage.com

Wikipedia

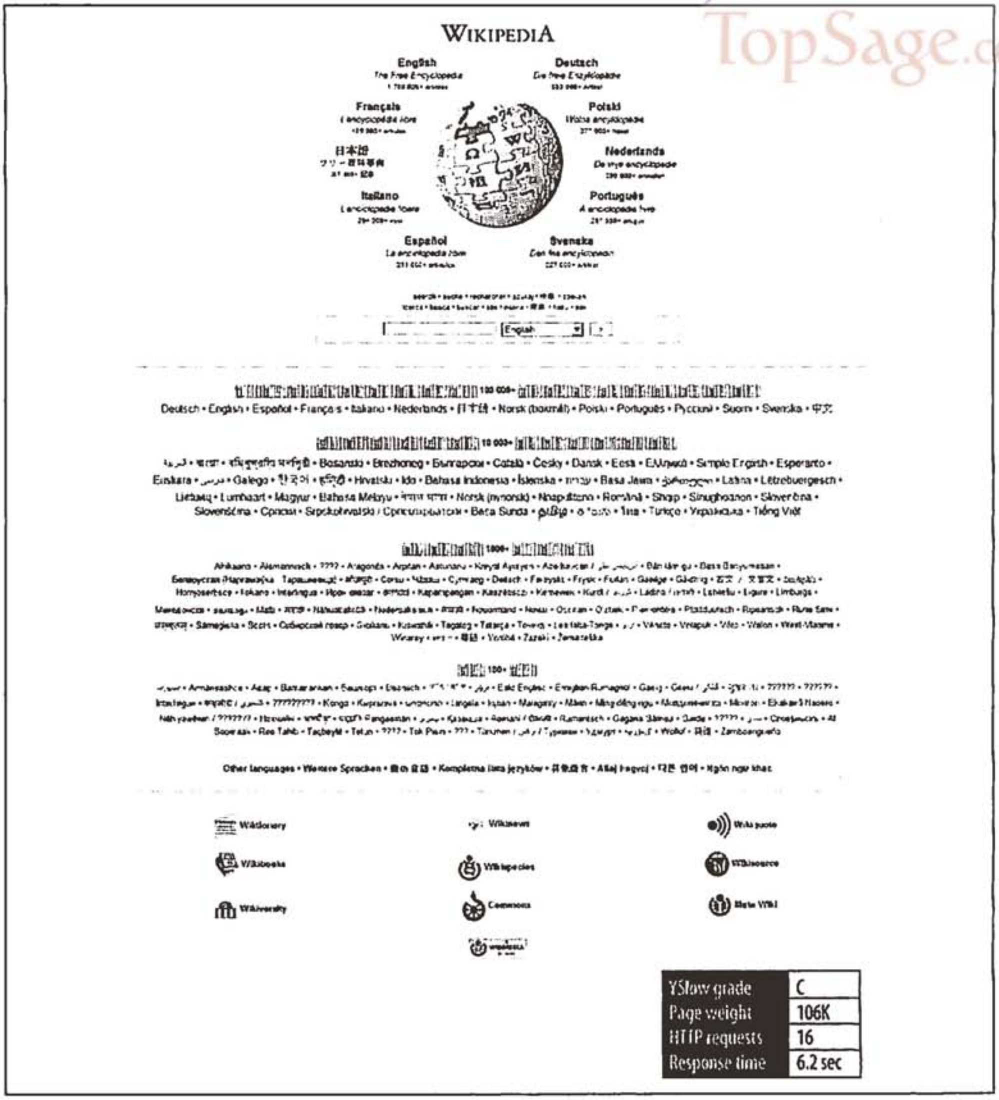


图 15-22：http://www.wikipedia.org

Wikipedia 页面（如图 15-22 所示）相对小而且快。如果将页面底部的 10 个用于导航的图片转换为 CSS Sprites，它还能更快。更进一步说，还可以合并两个样式表。这些简单的改进可以将页面的 HTTP 请求从 16 个减少到只有 6 个——HTML 文档、1 个样式表、3 个图片和 1 个 CSS Sprites。

www.TopSage.com

所有图片都没有 `Expires` 头。这是 Wikipedia 可以做出的第二重要的性能改进。其中有些图片超过一年也不会更改。添加一个长久的 `Expires` 头可以为数百万用户改善响应时间，以保证当图片改变时也不会给开发过程添加太多负担。

另外，样式表应该进行压缩。它们目前的总大小约 22KB，经过压缩可以将其字节数减小到 16KB。

Wikipedia 的大多数图片是 PNG 格式的。PNG 格式比起 GIF 通常是更好的选择，因为它的文件体积更小，并且具有更高的颜色深度和透明度。使用 PNG 格式可能已经为 Wikipedia 省下了数千字节的下载数据了（不大可能将其 PNG 图片转换成 GIF 来进行压缩，因为这样做会损失颜色深度）。然而，尽管已经选择了 PNG 格式，进一步的优化还是能将文件大小减少更多。例如，优化 Wikipedia 的 12 个 PNG 图片可以将整体大小从 33KB 减少到 28KB，节省了 15%。可供使用的 PNG 优化器有很多——我使用的是 PngOptimizer（*http://psydk.org/PngOptimizer.php*）。向开发过程中添加一个 PNG 优化步骤可以改善 Wikipedia 的性能。

www.TopSage.com

大家网 TopSage.com

Yahoo

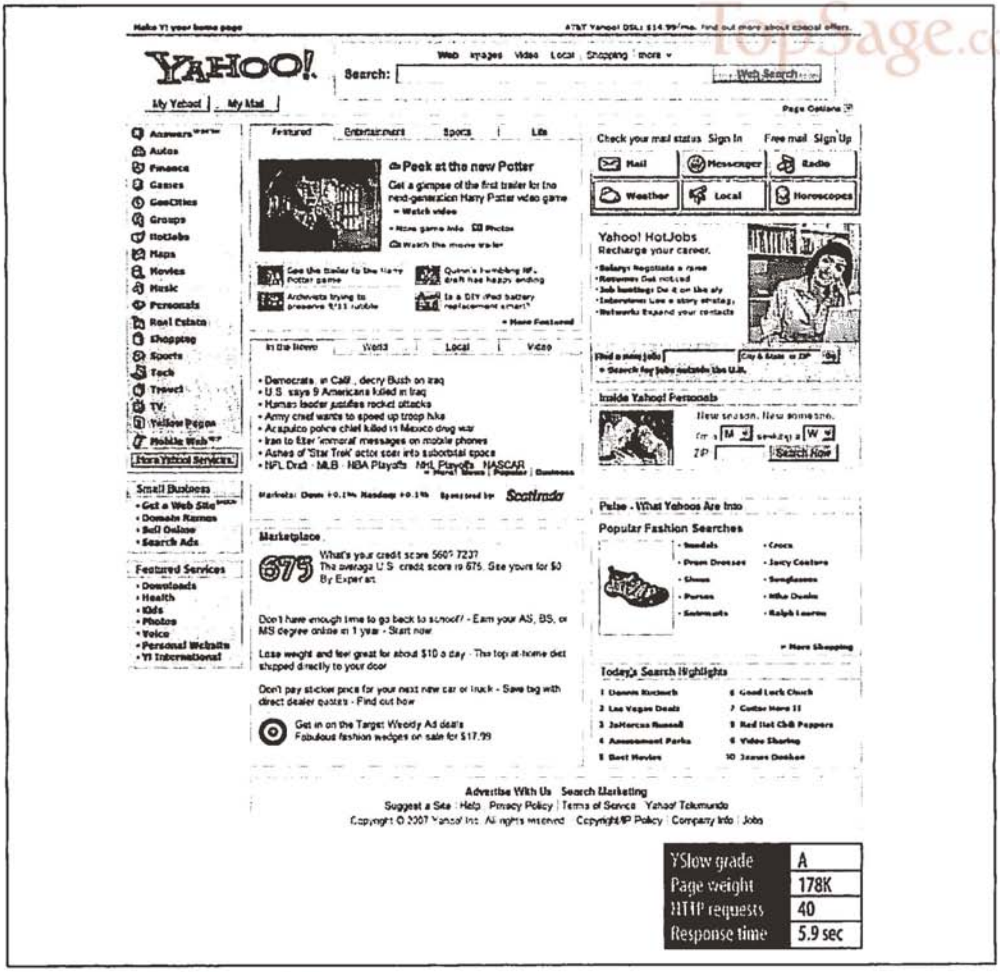


图 15-23：http://www.yahoo.com

Yahoo!（*http://www.yahoo.com*）的页面如图 15-23 所示，它的总字节数在本章检测的所有页面中位居第四，但其响应时间和 YSlow 等级排在第二。Yahoo!的主页团队和我的性能团队已经打了很多年交道了。因此，他们的 YSlow 得分很高，并且他们有能力从页面中压榨更多的速度出来。

Yahoo!主页有 4 个 CSS Sprites。它是已经使用了很多年的 CSS Sprites 了，Yahoo!也是我所遇到的第一个使用 CSS Sprites 的网站。其中一个 CSS Sprites 是 *icons_1.5.gif*。查看一下

www.TopSage.com

组件列表，可以发现该图片被下载了两次。经过进一步调查，发现其原因是两个 URL 引用了完全一样的图片——

http://us.js2.yimg.com/us.js.yimg.com/i/ww/sp/icons_1.5.gif
http://us.i1.yimg.com/us.yimg.com/i/ww/sp/icons_1.5.gif

这个错误是如何产生的呢？很可能是使用了模板变量来构建这些 URL。包含了该背景图片的 CSS 规则都是内联在 HTML 文档中的，可能是两个规则都访问了相同的模板变量。*us.js2.yimg.com* 这个主机名用于所有的脚本，而 *us.i1.yimg.com* 仅用于图片和 Flash。很可能在该 CSS 背景图中偶然使用了“JavaScript”主机名——*us.js2.yimg.com*。

从主机名的使用上可以看出 Yahoo!主页的一些良好的性能优化。他们将组件分开放到多个主机名下，从而增加了并行下载，如图 15-24 所示。此外，他们还选择了 *yimg.com* 这个域名，和页面的主机名 *yahoo.com* 不同。其结果是，针对 *yimg.com* 的 HTTP 请求中不会用到任何存在于 *yahoo.com* 域名下的 Cookie。当我登录到我的 Yahoo!个人账户后，*yahoo.com* 的 Cookie 超过了 600 字节，因此在页面的所有 HTTP 请求中，这将总共节省超过 25KB 的数据。

有两个元素的文件名比较有意思——*onload_1.3.4.css* 和 *onload_1.4.8.js*。在第 5 章和第 6 章中我谈到了样式表和脚本对性能的负面影响（样式表阻塞页面的呈现，脚本阻塞其后面元素的呈现和下载）。我在第 8 章中介绍了有关这一问题的优化，就是在页面加载完毕后再下载这些组件，从而消除负面的阻塞效果。这种极端的方式只适用于当初始页面的呈现不需要这些样式表或脚本时。在 Yahoo!主页中，该样式表和脚本仅用于页面加载之后的 DHTML 动作。例如，单击“More Yahoo! Services”链接会显示一个 DHTML 列表，其中有其他 Yahoo!功能的链接。这一功能在页面加载之后才会出现，它包含在 *onload_1.3.4.css* 中。

可以在 Yahoo!上进行的主要改进，不是移除前面介绍的重复 CSS 背景图片，而是减少域名的数量（7 个）和合并页面中的 3 个脚本。精简 HTML 文档（像 MSN 那样）可以将其从 117KB 减小到 29KB。整体来说，Yahoo!主页展示了很多高级的性能优化，而且对于页面中的内容和功能改进提供了最快的响应速度。

www.TopSage.com

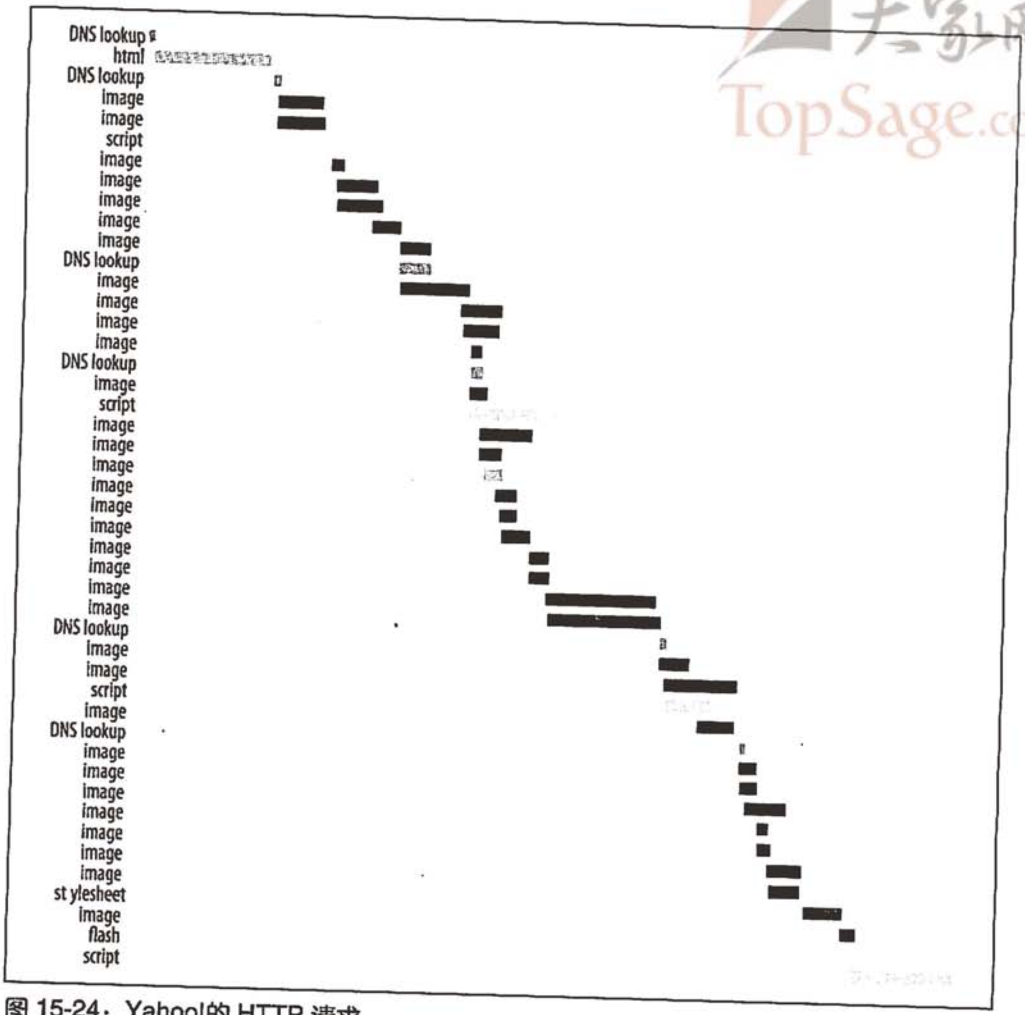


图 15-24：Yahoo!的 HTTP 请求

www.TopSage.com

大家网
TopSage.com

YouTube

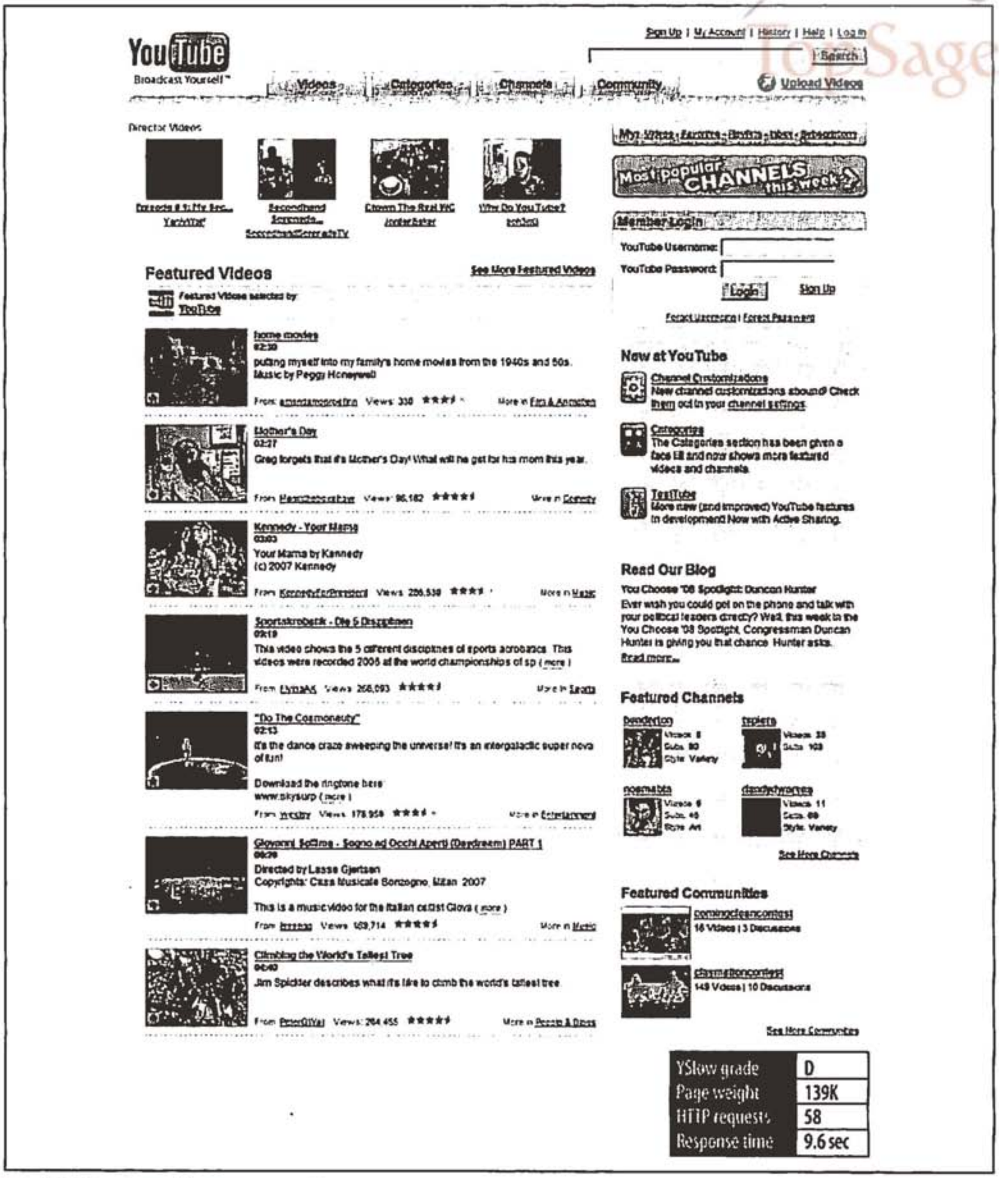


图 15-25：http://www.youtube.com

YouTube 的主页（*http://www.youtube.com*）（如图 15-25 所示）并不是很大，但它的 YSlow 等级很低，并且其后半部分的响应速度很慢。图 15-26 表明，其开始和结束时并没有很多的并行下载。增加这一部分的并行下载程度可以使响应速度产生巨大的改进。

www.TopSage.com

大家网
TopSage.com

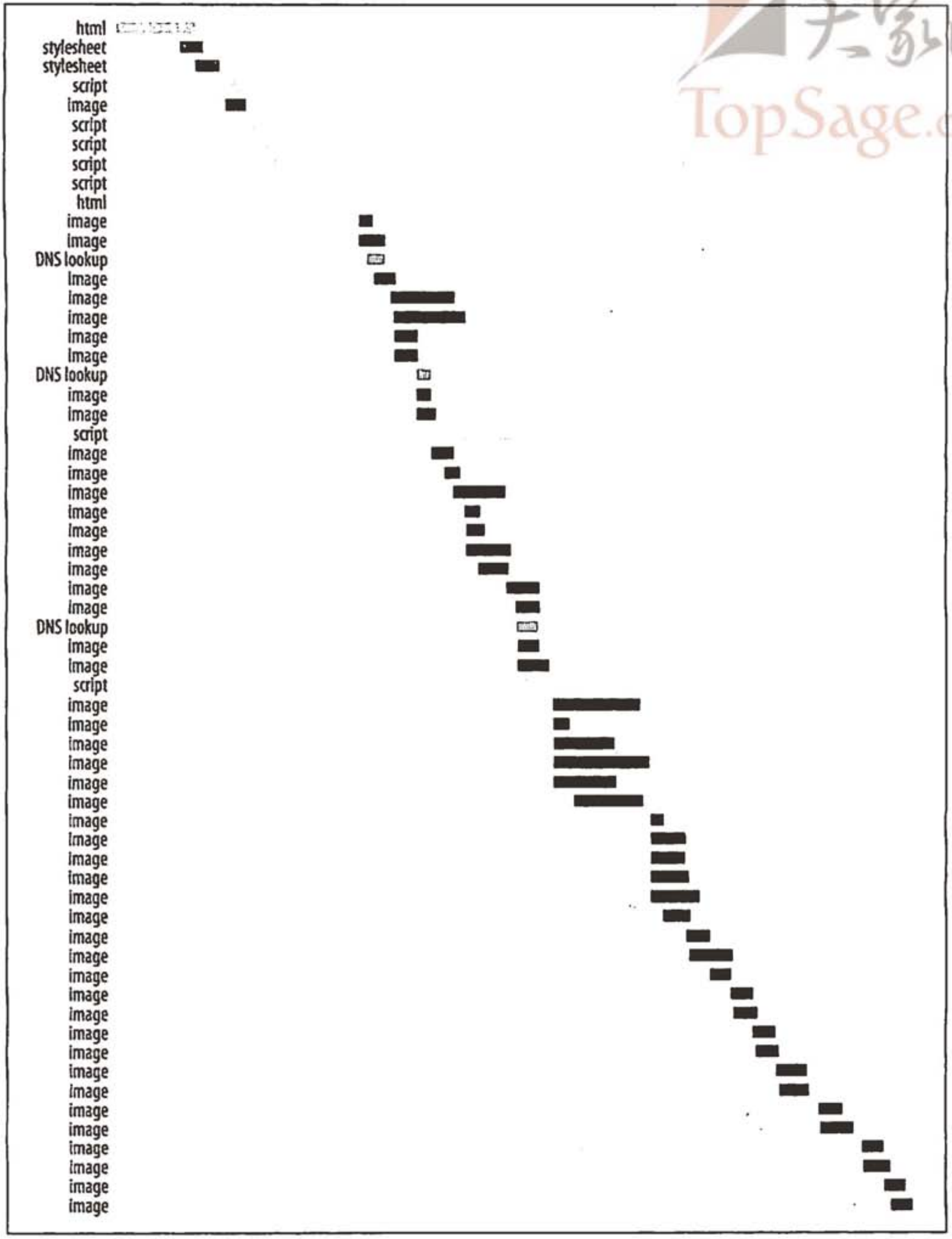


图 15-26：YouTube 的 HTTP 请求

www.TopSage.com

在页面加载开始时，并行下载的主要阻碍是 6 个连着下载的脚本。第 6 章曾解释过，不管其主机名是什么，脚本会阻塞其他所有的下载，且这些脚本没有经过精简。将这 6 个脚本合并为一个并进行精简，可以减少下载时间。此外，如果这些脚本能在页面中比较靠后的位置下载，页面的开始部分可以很快地下载和呈现。

在页面的结尾，并行下载数量的减少是因为从一个主机名（*img.youtube.com*）下载了 15 个图片。YouTube 只在页面中使用了 4 个唯一主机名。因此值得花费一次额外的 DNS 查找来将 15 次下载分别放到两个主机名下，这样能够使并行下载的数量加倍。

不幸的是，没有一个组件具有长久的 `Expires` 头（规则 3）。页面中绝大多数组件都是用户生成的图片，它们会频繁地滚动。为这些图片添加 `Expires` 头效果不大，单页面中的其他组件变化得不是这么频繁。其中 11 个组件 6 个月或更久都不会变化。为这些组件添加长久的 `Expires` 头可以缩减后续页面查看的响应时间。

YouTube 使用了 Apache Web 服务器，并且其组件包含 ETag，但 YouTube 进行了额外的工作，修改了 ETag 的语法，改善了它的可缓存性，如第 13 章所述。

www.TopSage.com

www.TopSage.com

索引

Index

数字

204 No Content 状态码

　　AOL，113

300 Multiple Choices（基于 Content-Type）状态码，76

301 Moved Permanently 状态码，76

302 Moved Temporarily（即 Found）状态码，76

303 See Other（对 302 的说明）状态码，76

304 Not Modified 状态码，76，90

304 响应，8

305 Use Proxy 状态码，76

306 状态码（不再使用），76

307 Temporary Redirect（对 302 的说明）状态码，76

A

Accept-Encoding，33

广告、服务，117

Ajax，96-102

　　主动请求，98

　　缓存示例，99-102

　　　　Google Docs & Spreadsheets，101

　　　　Yahoo! Mail，99-101

　　定义，97

　　长久的 Expires 头 102

　　优化请求，99

　　被动请求，98

　　Web 2.0、DHTML 和 Ajax 之间的关系，96

　　背后的技术，98

　　Yahoo! UI（YUI）Connection Manager for Ajax，98

Akamai Technologies, Inc.，19

Alias 指令，80、81

Amazon

　　CSS Sprites，108

　　Expires 头，107

　　下载时间的百分比，4

　　性能建议，107

　　（参见十大美国网站）

AOL

　　204 No Content 状态码，113

　　信标，113

　　DNS 查找，113

　　Expires 头，112

　　gzip，113

　　HTTP 请求，110

　　下载时间的百分比，4

　　性能建议，110-113

　　脚本，112

　　（参见十大美国网站）

Apache 1.3 mod_gzip 模块，31

Apache 2.x mod_deflate 模块，32

应用程序 Web 服务器、接近用户，18

自动索引，80

B

信标，82

　　AOL，113

　　警告，83

BrowserMatch 指令，34

浏览器、当其行为不同时，44

C

缓存、DNS，66

缓存、空缓存 VS 完整缓存，56

可缓存的外部 JS 和 CSS（示例），58

Cache-Control 头，23、25

　　max-age 指令，23

　　　　十大美国网站，24

CDN（示例），20

CDN（参见内容发布网络）

客户端图片地图，11

CNAMEs（DNS 别名），47、81

CNN

　　CSS Sprites，115

　　下载时间的百分比

　　性能建议，114-115

　　用图片显示文字，115

　　（参见十大美国网站）

CoDeeN，20

合并脚本（示例），16

www.TopSage.com

组件 Web 服务器、接近用户，18
组件
延迟，38
确保用户获取了最新版本，27
夸张的响应时间，39
修改 ETag 的示例，94
长久的 Expires 头，25-27
如何缓存和验证，89-91
条件 GET 请求，90
ETag，91
Expires 头，89
Last-Modified 响应头，90
重用，57
样式表（见样式表）
不必要的重新加载，92
服务器验证其是否与原始服务器上的组件相
匹配的方法，90
压缩
deflate（见 deflate）
边缘情形，34-35
如何工作，29
HTTP，7
HTTP 响应（见 gzip）
页面大小节省，36
使用了 deflate 和 gzip 的大小，31
压缩什么，30
条件 GET 请求，3、7、8、90
ETag，8
If-None-Match 头，8
内容发布网络（CDN），18-21
Akamai Technologies, Inc.，19
收益，20
CoDeeN，20
CoralCDN，20
定义，19
缺点，20
Globule，20
Limelight Networks, Inc.，19
Mirror Image Internet, Inc.，19
响应时间改善，20
SAVVIS Inc.，19
服务提供商，19
免费，20
十大美国网站，29
Speedera Networks，19
内容、地理上的分布，18
CoralCDN，20
Crockford、Douglas，70
CSS，55-62
合并，15-16
动态内联，60-62
示例
将 CSS 放在底部，39
将 CSS 放在顶部，41
使用@import 将 CSS 放在顶部，41
无样式内容的 CSS 闪烁，43
CSS Sprites，13
表达式计数器，52
表达式，51-54
事件处理器，53
一次性表达式，52
避免问题的技术，52
更新，52
什么导致性能变差，51
主页，58
内联 VS 外部，55-58
组件重用，57
空缓存 VS 完整缓存，56
内联示例，55
页面查看，56
权衡，58
精简，75
加载后下载，59
Sprites，11-13
Amazon，108
CNN，115
Google，121
MSN，124
Wikipedia，130
Yahoo!，132

D

data: URL 模式，13
主要缺点，14
延迟脚本（示例）50
deflate，30
压缩大小，31
延迟组建，38
DELETE 请求，6
DHTML
定义，97
Web 2.0、DHTML 和 Ajax 之间的关系，96
DirectorySlash，80、81

www.TopSage.com

大家网 TopSage.com

DNS（Doain Name Service）
别名，47
浏览器白名单方式，34
缓存，66
角色，63
DNS 查找，63-68
AOL，113
浏览器的视角，66-68
Firefox，67
Internet Explorer，66
缓存和 TTL，63-66
十大美国网站发送到客户端的最大 TTL 值，65
eBay，117
影响缓存的因素，64
Keep-Alive，67、68
MSN，124
MySpace，128
减少，68
Dojo Compressor，70
gzip 压缩后减少的大小，74
使用之后减少的大小，71
Domain Name System（见 DNS；DNS 查找）
下载
并行，46-48
开销，47
限制，46
脚本阻塞，48
重复脚本——10 次缓存（示例），87
重复脚本——缓存（示例），86
重复脚本——无缓存（示例），86
动态内联（示例），61

E

eBay
DNS 查找，117
ETag，117
Expires 头，117
HTTP 请求 117
IFrame，117
图片，117
下载时间的百分比，4
性能建议，116-119
脚本，119
（参见十大美国网站）
实体标签（见 ETag）
ETag，35、89-95
条件 GET 请求，8
定义，89
eBay，117
代理缓存的效果，92
改变了 ETag 的组件示例，94
Apache 1.3 和 2.x 的格式，92
IIS 的格式，92
MSN，124
选项，93
问题，91
移除，93
十大美国网站，94-95
YouTube，137
事件处理器，53
示例，53
压缩所有组件（示例），35
示例
可缓存的外部 JS 和 CSS，58
CDN，20
合并脚本，16
将 CSS 放在底部，39
将 CSS 放在顶部，41
使用@import 将 CSS 放在顶部，41
无样式内容的 CSS 闪烁，43
CSS Sprites，13
延迟脚本，50
重复脚本——10 次缓存，87
重复脚本——缓存，86
重复脚本——无缓存，86
动态内联，61
事件处理器，53
压缩所有内容，35
表达式计数器，52
外部 JS 和 CSS，55
长久的 Expires，28
HTML 压缩，35
图片信标，83
图片地图，11
内联 CSS 图片，14
内联图片，14
内联 JS 和 CSS，55
经过精简的大脚本，72
一般的大脚本，72
经过混淆的大脚本，72
无 CDN，20
无 Expires，28

www.TopSage.com

无图片地图，11
无压缩，35
一次性表达式，53
加载后下载，59
将脚本放在底部，50
将脚本放在顶部，49
脚本阻塞下载，48
将脚本放在中部，45
顶部脚本 VS 底部脚本，50
分离的脚本，16
经过精简的小脚本，72
一般的小脚本，72
经过混淆的小脚本，72
在哪里查找在线示例，xv
XMLHttpRequest 信标，83
Expires 头，8、22-28、89
选择，23
Amazon，107
AOL，112
组件
确保用户获取了最新版本，27
十大美国网站，26
定义，22
eBay，117
空缓存 VS 完整缓存，24
mod_expires，23
MSN，124
MySpace，128
十大美国网站，24
Wikipedia，131
YouTube，137
（参见长久的 Expires 头）
表达式计数器（示例），52
表达式方法（参见 CSS，表达式）
外部 JS 和 CSS（示例），55

F

长久的 Expires 头（示例），28
长久的 Expires 头，25、100
Ajax，102
缓存，28
组件，25-27
定义，23
示例，28
页面查看，24
Fasterfox，68
favicon，120
file_get_contents PHP 函数，14
fileETag 指令，93
Firebug，106
Firefox
延迟脚本，50
DNS 查找，67
重复脚本，86
并行下载，46
管道，9
逐步呈现，14
前端性能，1-5

G

Garrett、Jesse James，97、98
地理上分布的内容，18
GET 请求，6
条件（见条件 GET 请求）
Globule，20
Gomez，21
Web 监视服务，105
Google
CSS SPRITES，121
HTTP 请求，120
下载时间的百分比，4
性能建议，120-122
SCRIPT DEFER 属性，122
（参见十大美国网站）
Google Docs & Spreadsheets，101
Google Toolbar、重定向，84
gzip，29-36
AOL，113
命令行工具，32
压缩
边缘情形，34-35
压缩大小，31
配置
Apache 1.3 mod_gzip 模块，31
Apache 2.x mod_deflate 模块，32
示例，35
压缩是如何工作的，29
图片和 PDF 文件，30
精简，74
mod_gzip 文档，34
MSN，124
IE 中的问题，34
代理缓存，33

大家网 TopSage.com
www.TopSage.com

十大美国网站，30
压缩什么，30
Wikipedia，131

H

HEAD 请求，6
Hewitt、Joe，106
主页，58
主机名、减少，68
HTML 压缩（示例），35
HTTP
304 请求，8
压缩，7
Expires 头，8
GET 请求，6
条件，7
GET 请求，
条件，8
Keep-Alive，8
概述，6-9
持久连接，8
管道，9
响应、压缩（见 gzip）
规范，6、9
流量，3
HTTP 请求，10-17
AOL，110
CSS Sprites，11-13
eBay，117
Google，120
图片地图，10
客户端，11
缺点，11
服务器端，11
内联图片，13-15
JavaScript 和 CSS 合并，15-16
MSN，123、124
MySpace，128
加载后下载技术，16
类型，6
Yahoo!，133
http:模式，13
Hyatt、David，43

I

IBM Page Detailer，105
If-None-Match 头，8
IFrame
eBay，117
MSN，124
图片信标（示例），83
图片地图（示例），11
图片地图，10
客户端，11
缺点，11
服务器端，11
图片
缓存和不缓存，3
eBay，117
压缩，30
内联，13-15
内联 CSS 图片（示例），14
内联 JS 和 CSS（示例），55
inode，92
国际化，115
Internet Explorer
data:模式，14
延迟脚本，50
DNS 查找，66
重复脚本，87
gzip 缺陷，34
并行下载，46
管道，9
gzip 带来的问题，34
逐步呈现，43
XMLHTTP，98

J

JavaScript，55-62
合并，15-16
调试代码工具，106
依赖和版本，87
重复脚本，85-88
避免，87
性能，86
动态内联，60-62
主页，58
内联脚本
精简，73
内联 VS 外部，55-58
组件重用，57
空缓存 VS 完整缓存，56
内联示例，55
页面查看，56

www.TopSage.com

权衡，58
精简，69-75
定义，69
示例，72
MSN，124
MySpace，128
节省，70-72
混淆，70
加载后下载，59
脚本管理模块，87
榨取浪费的时间，73-75
（参见脚本）
JSLint，106
JSMin，70
gzip 压缩后减少的大小，74
使用，71

K

Keep-Alive，8
DNS 查找，67、68
Firefox VS IE，67
Keynote Systems，21

L

经过精简的大脚本（示例），72
一般的大脚本（示例），72
经过混淆的大脚本（示例），72
Last-Modified 日期，26
Last-Modified 头，26
Last-Modified 响应头，90
Limelight Networks, Inc.，19

M

max-age 指令，23
十大美国网站，24
精简
定义，69
JavaScript（见 JavaScript，精简）
十大美国网站，69
Mirror Image Internet, Inc.，19
mod_autoindex，80
mod_deflate 模块，32
mod_dir，80
mod_expires，23
mod_gzip 文档，34
mod_gzip 模块，31
mod_gzip_minimum_file_size 指令，30
mod_rewrite 模块，80
MSN
CSS Sprites，124
DNS 查找，124
ETag，124
Expires 头，124
gzip，124
HTTP 请求，123、124
IFrame，124
JavaScript 精简，124
下载时间的百分比，4
性能建议，123-125
（参见十大美国网站）
MySpace
DNS 查找，128
Expires 头，128
HTTP 请求，128
JavaScript 精简，128
下载时间的百分比，4
性能建议，127-128
（参见十大美国网站）

N

network.http.max-persistent-connections-per-server 设置，46
New York University，20
Nielson、Jakob，38
无 CDN（示例），20
无压缩（示例），35
无 Expires（示例），28
无图片地图（示例），11
Nottingham、Mark，27

O

O'Reilly、Tim，97
混淆，70
一次性表达式（示例），53
优化选择，70
OPTIONS 请求，6

P

页面查看，56
页面大小、十大美国网站，103
并行下载，46-48
开销，47
限制，46
并行，112
YouTube，137
被动请求，98

www.TopSage.com

PDF 文件、压缩，30
性能
缓存和无缓存图片，3
条件 GET 请求，3
指出时间花在哪里，3
前端，1-5
十大美国网站在下载时花费的时间百分比，4
剖析，4
建议
Amazon，107
AOL，110-113
CNN，114-115
eBay，116-119
Google，120-122
MSN，123-125
MySpace，127-128
Wikipedia，130-131
Yahoo!，132-133
YouTube，135-137
重定向，3
通过 CDN 得到的响应时间改善，20
响应时间测试，21
脚本，3
十大美国网站概述，103
十大美国网站
如何完成测试，105
跟踪 Web 页面，1
性能黄金法则，4、5
持久连接，8
管道，9
PlanetLab，20
PNG 图片，131
POST 请求，6
加载后下载（示例），59
加载后下载技术，16
预加载，121
Princeton University，20
逐步呈现，37
代理缓存
gzip，33
PUT 请求，6

R
重定向，3、76-84
十大美国网站中的，79
选择，79-84
连接网站，81
缺少结尾的斜线，79
美化 URL，84
跟踪内部流量，81
跟踪出站流量，82-84
性能是如何被损伤的，77-79
类型，76
呈现、逐步，27
响应时间
最大的影响，46
让 HTTP 响应离用户更近（见内容发布网络）
省掉不必要的 HTTP 请求（见 Expires 头）
减小 HTTP 响应的大小（见 gzip）
测试，21
十大美国网站，103

S
SAVVIS Inc.，19
模式，13
SCRIPT DEFER 属性（Google），122
脚本，3、45-50
AOL，112
放在页面底部，49
放在页面顶部，49
阻塞下载，48
延迟，50
依赖和版本，87
重复，85-88
避免，87
性能，86
eBay，119
十大美国网站中的数量，85
并行下载，46-48
问题，45
脚本管理模块，87
Yahoo!，133
（参见 JavaScript）
将脚本放在底部（示例），50
将脚本放在顶部（示例），49
脚本阻塞下载（示例），48
将脚本放在中部（示例），45
顶部脚本 VS 底部脚本（示例），50
分离脚本（示例），16
ServerInfoTimeOut 值，67
服务器端图片地图，11
Shea、Dave，13

www.TopSage.com

ShrinkSafe，70
sleep.cgi，38
经过精简的小脚本（示例），72
一般的小脚本（示例），72
经过混淆的小脚本（示例），72
Speedera Networks，19
样式表，37-44
白屏，39-42
避免，43
样式表放在底部和放在顶部的示例，39-42
将 CSS 放在底部，39
将 CSS 放在顶部，41-42
无样式内容的闪烁，43
避免，43
十大美国网站中的数量，85
将其放在文档底部的问题，38

T

将文字做成图片，115
Theurer、Tenni，25
十大美国网站
CDN 服务提供商，19
带有 Expires 头的组件，26
ETag，94-95
Expires 头和 max-age 指令，24
gzip 的使用，30
性能测试如何完成，105
发送到客户端的最大 TTLL 值，65
精简，69
精简内联脚本，73
脚本和样式表的数量，85
页面大小，103
下载时间所占的百分比，4
性能概述，103
重定向，79
响应时间，103
脚本和样式表，15
YSlow 等级，103-105
TRACE 请求，6
TTL
DNS 缓存，63-66
十大美国网站发送到客户端的最大 TTL 值，65

U

URL、美化，84

V

可视化反馈，37
Vrije Universiteit，20

W

Web 2.0，96-102
定义，97
Web 2.0、DHTML 和 Ajax 之间的关系，96
Web 页面性能，1
Wikipedia
CSS Sprites，130
Expires 头，131
gzip，131
下载时间的百分比，4
性能建议，130-131
PNG 图片，131
（参见十大美国网站）

X

XMLHttpRequest，83
XMLHttpRequest 信标（示例），83

Y

Yahoo!，1、4
CSS Sprites，132
域名，133
HTTP 请求，133
下载时间的百分比，4
性能建议，132-133
脚本，133
两个 URL 引用了相同的图片，133
（参见十大美国网站）
Yahoo! Mail
Ajax 缓存示例，99-101
Yahoo! Search，4
Yahoo! Shopping 和 Akamai 的 CDN，21
Yahoo! UI（YUI）Connection Manager for Ajax，98
YouTube
ETag，137
Expires 头，137
并行，137
下载时间的百分比，4
性能建议，135-137
（参见十大美国网站）
YSlow，106
等级
定义，104
十大美国网站，103-105

www.TopSage.com

关于作者

大家网 TopSage.com

Steve Souders 在 Yahoo!担任 Chief Performance。他于 2000 年加盟 Yahoo!，在该公司的很多平台和产品团队中工作过。在他到达今天这个位置之前，他就职于 My Yahoo!开发团队。

作为 Chief Performance Yahoo!，他开发了一系列优秀软件，可以使网站访问速度变得更快。他构建了用于进行性能分析的工具，并将这些优秀软件和工具传播到 Yahoo!的各个产品团队中。

在到 Yahoo!之前，Steve 就职于很多小型或中型公司，包括他和别人一起创办的两个公司——Helix Systems 和 CoolSync。他还曾就职于 General Magic、WhoWhere?和 Lycos。在 20 世纪 80 年代早期，Steve 捕获到了 Artificial Intelligence 的 bug，并在一些公司里进行机器方面的研究。他在维吉尼亚大学得到了系统工程学学士学位，后又在斯坦福大学获得了管理科学和工程学硕士学位。

Steve 的兴趣爱好多种多样。他常出现在 Freehand System 的会议桌上，并且经常光顾 Fremont Hills Country Club，还在 Sunday School 教书。他曾与很多 NBA 和 WNBA 球员一起打篮球，但最近他改玩 Ultimate Frisbee 了。他是 Universal Studios Internet Task Force 的成员、他重建了有 90 年历史的车房（carriage house），他还参与了一项吉尼斯世界纪录。他有一位美丽的妻子和三个女儿。

关于封面

本书封面上的动物是灵缇犬（Greyhound，一种善跑的狗）。

作为世界上跑得最快的狗，灵缇犬的速度可以达到每小时 45 英里（约 72 千米），这源自它流线型、狭长的身躯，巨大的肺、心和肌肉，双倍的悬空飞跃（每一步都有两次四脚同时离地）及其灵活的脊柱。尽管灵缇犬跑得出人意料地快，但它的精力并不旺盛，而且缺乏耐力，因此它需要比其他的狗更多的锻炼时间。出于这一原因，它们经常被称作“45 英里每小时的懒汉”，因为除了追击一些小的猎物（如兔子和猫），它们会很知足地终日大睡。

灵缇是最古老的犬种之一，出现在各个历史时期的艺术和文学作品中。在古埃及，灵缇犬通常和他的主人一起制成木乃伊并埋葬，公元前 4000 年犬的象形文字和现代的灵缇犬非常像。在希腊和罗马，灵缇犬通常和上帝及女神一同描述。灵缇犬曾出现在霍默、乔叟、莎士比亚和塞万提斯的作品中，而且它们还是《圣经》中唯一提到的一种狗。它们卓越的智商、优雅的外形、优秀的竞技能力和忠实的态度长久以来一直为人们所津津乐道。在

www.TopSage.com

20 世纪 20 年代早期，现代灵缇犬竞技被引入美国。参加竞技的灵缇犬比展示犬更小而且更耀眼，它们经过有选择地繁衍，通常保持身长 635 毫米~736 毫米（25 英寸~29 英寸）、体重 27 千克~32 千克（60 英镑~70 英镑）。这些狗可以追上任何快速移动的东西（作为观赏犬，而不是侦察犬），因此它们得以在赛道中追逐它们的诱饵——机械野兔。灵缇犬竞赛在美国仍然是一种非常流行的观赏性比赛，和赛马一样，是一种颇受喜爱的博彩形式。

但灵缇犬竞赛非常受争议，因为狗们体验到的人类关怀太少了，而且它们的非竞赛时间大多花来配种。一旦灵缇犬衰老不适宜比赛（一般在 3 岁到 5 岁之间），很多都被安乐死了，因此产生了很多营救计划，为这些退休的选手寻找新家。灵缇犬本性温顺、冷静，很多能经过驯养而变成可爱的宠物。

www.TopSage.com

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java一览无余：Java视频教程 | Java SE | Java EE

.Net技术精品资料下载汇总：ASP.NET篇

.Net技术精品资料下载汇总：C#语言篇

.Net技术精品资料下载汇总：VB.NET篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI脚本语言编程学习资源下载地址大全

Python语言编程学习资料(电子书+视频教程)下载汇总

最新最全Ruby、Ruby on Rails精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL篇 | SQL Server篇 | Oracle篇

平面设计优秀资源学习下载 | Flash优秀资源学习下载 | 3D动画优秀资源学习下载

最强HTML/xHTML、CSS精品学习资料下载汇总

最新JavaScript、Ajax典藏级学习资料下载分类汇总

网络最强PHP开发工具+电子书+视频教程等资料下载汇总

UML学习电子资下载汇总 软件设计与开发人员必备

经典LinuxCBT视频教程系列 Linux快速学习视频教程一帖通

天罗地网：精品Linux学习资料大收集(电子书+视频教程) Linux参考资源大系

Linux系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD精品学习资源索引 含书籍+视频

Solaris/OpenSolaris电子书、视频等精华资料下载索引